MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

WILLIAM TAVARES DE LIMA LADEIRA

UMA ANÁLISE DAS TÉCNICAS DO AES EM WHITE-BOX E SUA APLICAÇÃO EM DOIS ESTUDOS DE CASO

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA

WILLIAM TAVARES DE LIMA LADEIRA

UMA ANÁLISE DAS TÉCNICAS DO AES EM WHITE-BOX E SUA APLICAÇÃO EM DOIS ESTUDOS DE CASO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. José Antônio Moreira Xexéo - D.Sc.

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

005.1 Ladeira, William Tavares de Lima
L153a Uma análise das técnicas do AES em White-box e sua aplicação em dois estudos de caso / William Tavares de Lima Ladeira, orientado por José Antônio Moreira Xexéo - Rio de Janeiro: Instituto Militar de Engenharia, 2017.

123p.: il.

Dissertação (mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2017.

1. Curso de Sistemas e Computação - teses e dissertações. 1. White-box. 2. Twofish. 3. AES. 4. Rijndael. 5. Black-box. 6. Gray-box. 7. MDS. I. Xexéo, José Antônio Moreira . II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

WILLIAM TAVARES DE LIMA LADEIRA

UMA ANÁLISE DAS TÉCNICAS DO AES EM WHITE-BOX E SUA APLICAÇÃO EM DOIS ESTUDOS DE CASO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. José Antônio Moreira Xexéo - D.Sc.

Aprovada em 11 de outubro de 2017 pela seguinte Banca Examinadora:

Prof. José Antônio Moreira Xexéo - D.Sc. do IME - Presidente

Prof. Anderson Fernandes Pereira dos Santos - D.Sc. do IME

Prof. Flávio Luis de Mello - D.Sc. da UFRJ

William Augusto R. de Souza - Ph.D. do Royal Holloway University of London

Rio de Janeiro
2017

Ao IME, à minha família, professores e amigos.

AGRADECIMENTOS

Agradeço primeiramente à minha família, que sempre confiou na minha capacidade, ao professor Matheus Bandini, fundamental para o meu ingresso no mestrado e para que hoje eu ambicione o doutorado, ao professor Douglas Ericson, que também acreditou no meu potencial e me permitiu alcançar essa nova etapa.

Aos professores do IME em geral, com os quais pude evoluir meu aprendizado e assumir desafios que excediam minhas expectativas. Dentre os professores, agradeço em especial ao meu orientador, Dr. José Antônio Moreira Xexéo que, através de seus questionamentos e críticas, permitiu a este trabalho ser o que ele é.

Aos colegas de pesquisa, Yuri Waki, Eduardo Marsola e Leandro, que forneceram dicas e compartilharam conhecimentos que contribuíram para a evolução desta dissertação.

Aos amigos, Lauro Víctor Ramos Cavadas e Victor Dias de Oliveira, fundamentais em diversas etapas cruciais do curso, com os quais tive a oportunidade de compartilhar as dificuldades e os sucessos gerados por esse novo desafio.

Aos meus colegas de trabalho, que me ofereceram suporte em situações em que conciliar o trabalho e o mestrado parecia tarefa impossível.

"Our work is never over"

DAFT PUNK

SUMÁRIO

LISTA DE ILUSTRAÇÕES 10

LISTA DE TABELAS 13

LISTA DE SIGLAS 14

1 INTRODUÇÃO 17
1.1 Motivação 17
1.2 Os ambientes de acesso 19
1.2.1 O ambiente *black-box* 19
1.2.2 O ambiente *gray-box* 19
1.2.3 O ambiente *white-box* 20
1.3 Objetivos de pesquisa 21
1.4 Problemas de pesquisa 21
1.5 Organização do trabalho 21

2 TABELAS DE PESQUISA 23
2.1 As funções do AES 24
2.1.1 *ShiftRows* 24
2.1.2 *SubBytes* 25
2.1.3 *MixColumns* 25
2.1.4 *AddRoundKeys* 26
2.1.5 Execução convencional do AES 27
2.2 Reorganizando as funções do AES para a criação das tabelas 27
2.3 Analisando a estrutura do algoritmo para adaptá-lo ao ambiente *white-box* . . 28
2.4 Remodelando as funções em tabelas de pesquisa 30
2.4.1 *TBoxes* 30
2.4.2 *TyiTables* 32
2.4.3 *XORTables* 33
2.4.4 *TBoxesTyiTables* 34
2.4.5 Sobre aglutinar as *TBoxes* com as *TyiTables* 34
2.4.6 Resultado 34

3 CONFUSÃO 36
3.1 A confusão segundo Shannon 36
3.1.1 O método da palavra provável 37
3.2 A utilização da codificação no WBAES 38
3.2.1 Introduzindo as codificações entre as tabelas de pesquisa 38
3.2.2 Codificações internas 39
3.2.2.1 Segurança local 41
3.2.3 Codificações externas 42
3.3 Uma *sbox* com valores dependentes da chave 43
3.3.1 A utilização de elementos do Twofish em *white-box* 44

4 DIFUSÃO 46
4.1 Funções responsáveis por introduzir difusão 46
4.2 A matriz MDS 47
4.2.1 Matrizes de Cauchy 48
4.3 Bijeções misturadoras 49
4.3.1 Definindo as bijeções misturadoras 50
4.3.2 O funcionamento das bijeções 51

5 A CHAVE CRIPTOGRÁFICA 55
5.1 A utilização de funções *one-way hash* para expansão da chave 55
5.1.1 O Scrypt 56
5.2 As propostas de Klinec x As implementações de Bačinská - A chave expandida para as diferentes funções em *white-box* 57
5.2.1 Chaves de *round* 58
5.2.2 Uma *sbox* com dependência de chave 59
5.2.3 O uso de matrizes MDS com valores dinâmicos 60
5.2.3.1 Geração das matrizes MDS para um único *round* 61

6 RECOMENDAÇÕES *WHITE-BOX* E SUAS APLICAÇÕES PARA A CRIAÇÃO DO WBTWOFISH E WBTWOFISH+ 64
6.1 Funções não-lineares 65
6.2 Funções lineares 66
6.3 Utilizando funções *one-way hash* na manipulação da chave criptográfica 68
6.4 A escolha de um algoritmo como estudo de caso 68
6.5 O Twofish e sua estrutura 69

6.6 Uma implementação *white-box* para o Twofish 69
6.6.1 As implicações das técnicas de *input* e *output Whitening* em um contexto *white-box* 69
6.6.2 A criação do WBTwofish: Uma versão do Twofish que não altera criptograma 70
6.6.2.1 Alterando a organização dos rounds do WBTwofish 71
6.6.2.2 A estrutura do WBTwofish de acordo com a rede de Feistel 72
6.6.2.3 Modificando o funcionamento dos dois primeiros *rounds* para minimizar o vazamento da chave 73
6.6.3 O funcionamento do WBtwofish para os demais *rounds* e o seu comportamento durante a decriptação 77
6.6.4 WBTwofish: Mais seguro para o ambiente *white-box* 77
6.7 WBTwofish+: Uma nova versão do Twofish para o contexto *white-box* 78
6.8 Experimentos e resultados 79

7 CONSIDERAÇÕES FINAIS 81
7.1 Contribuições 81
7.2 Trabalhos futuros 82

8 REFERÊNCIAS BIBLIOGRÁFICAS 83

9 APÊNDICES 86
9.1 APÊNDICE 1: Classes compartilhadas - WBTwofish e WBTwofish+ 87
9.2 APÊNDICE 2: Geração de chaves e SBox - WBTwofish 118
9.3 APÊNDICE 3: Geração de chaves e SBox - WBTwofish+ 121

LISTA DE ILUSTRAÇÕES

FIG.1.1 Demonstração dos privilégios de um atacante em um ambiente *white-box*, em contrapartida aos recursos oferecidos em um ambiente *black-box*. Enquanto no ambiente *black-box*, o atacante possui acesso apenas ao texto de entrada e de saída do algoritmo, sem visibilidade da execução, no ambiente *white-box*, esse mesmo atacante acessa toda a implementação do software e possui controle sobre o ambiente de execução. Ilustração modificada de Lepoint et al. (2013) 18

FIG.2.1 Representação da função *ShiftRows*, permutando os *bytes* de acordo com os índices r e c, que indicam respectivamente linha e coluna da matriz a ser alterada. Ilustração extraída de Standard (2001) 25

FIG.2.2 ShiftRows, deslocando as três últimas linhas do State; extraída de Standard (2001). 25

FIG.2.3 *Sbox*: Matriz de substituição com valores em hexadecimal; extraída de Standard (2001). 26

FIG.2.4 Multiplicação realizada entre os *bytes* de uma determinada coluna do *state* e da matriz de 128 *bits* do *MixColumns*. Ilustração extraída de Standard (2001) 26

FIG.2.5 Representação de um *round* do AES-128 em formato de tabela de pesquisa, *round* 1. Funcionamento ilustrado por Muir (2013). 31

FIG.2.6 Multiplicação de *bytes* do *state* pela matriz *MC*. Ilustrado por Muir (2012) 32

FIG.2.7 Representação da multiplicação dos *bytes* do *state* pela matriz *MC* em vetor de coluna; resultando nas Ty_i*Tables* correspondentes a cada índice. 33

FIG.2.8 Ou-exclusivo de duas tabelas Ty_i. Para realizar essa operação foram necessárias 8 tabelas XOR, que combinaram os 64 *bits* de saída de duas Ty_i*Tables*. 33

FIG.2.9 Implementação do AES-128 em um contexto *white-box* após a aplicação das tabelas de pesquisa. Ilustrado por Muir (2013). 35

FIG.3.1 *TBox*Ty_i*Table*, que mapeia 8 *bits* para 32 *bits*. Ilustração extraída de Muir (2013) 39

FIG.3.2 Um elemento do conjunto A, que apresenta um correspondente diferente no conjunto B a cada nova codificação que o modifica. 42
FIG.3.3 Um elemento do conjunto B pode ser a correspondência de todos os elementos do conjunto A, que de acordo com suas respectivas codificações, podem apresentar o mesmo valor. 42

FIG.4.1 A alteração de um *byte* na entrada, afeta um única coluna na saída, devido ao funcionamento do *MixColumns*. 47
FIG.4.2 Matriz MixColumns que, apesar de MDS, não é involutória, pois não retorna a identidade quando multiplicada por si. 48
FIG.4.3 Matriz MDS 16x16, apresentada por Abrahao (2007) 49
FIG.4.4 Fluxo de dados para o round 1 do AES, com o *state* de entrada ao topo e o de saída na parte inferior. Ilustrado por Muir (2013) 52
FIG.4.5 Decomposição da bijeção misturadora em quatro tabelas de 8x32 *bits*. Ilustrado por Muir (2013) 52
FIG.4.6 Aplicação das bijeções misturadoras no *round* 2. A representação do *round* 3 ao 9 é a mesma. Para os rounds 1 e 10, a única diferença é a ausência de bijeções misturadoras na entrada das *Tboxes* para o *round* 1 e na saída das *Tboxes* para o *round* 10. Ilustrado por Muir (2013). 54

FIG.5.1 Expansão de chave para encriptar os rounds do algoritmo, proposta por Klinec et al. (2013). 58
FIG.5.2 Funcionamento da *sboxgen*. Proposta por Klinec et al. (2013). Ilustração extraída de Bačinská (2015) 60
FIG.5.3 Expansão de chave para a *sbox*. Proposta por Klinec et al. (2013), figura de Bačinská (2015) 60
FIG.5.4 Expansão de chave para as matrizes MDS. Proposta por Klinec et al. (2013), figura de Bačinská (2015) 61
FIG.5.5 Os seis primeiros e os seis últimos *bits* de cada *byte* de K_r são armazenados em um *Set*, que armazena apenas valores únicos. 62
FIG.5.6 Os primeiros 14 blocos de 6 *bits* do *Set*, são os primeiros 14 *bytes* da primeira linha. 62

FIG.6.1 Ilustração de uma forma de se alcançar o efeito avalanche em um único *round*, através de uma relação em que a matriz MDS possui,

pelo menos, o quadrado de *bits* do *state* 67
FIG.6.2 Twofish projetado por Schneier et al. (1998). 70
FIG.6.3 WBTwofish: Alteração na distribuição dos *rounds* do Twofish. 72
FIG.6.4 Simplificação do funcionamento do primeiro *round* do Twofish implementando a rede de Feistel. 73
FIG.6.5 Uma visão de um único round da função F (chave de 128 *bits*), extraído de Schneier et al. (1998). 75

LISTA DE TABELAS

TAB.6.1 Comparação dos algoritmos contra o BGE-Attack 80

LISTA DE SIGLAS

AES	Advanced Encryption Standard
WBAES	White-box AES
DES	Data Encryption Standard
NIST	National Institute of Standards and Technology
MDS	Maximum Distance Separable
WBTWofish	White-box Twofish
MB	Mixing Bijections
KDF	Key Derivation Function
PHT	Pseudo-Hadamard Transform
WB	White-box
DRM	Digital Rights Management
Tb	Tamanho do bloco
GMC	Gerador de Matrizes de Cauchy
GF	Galois Field

RESUMO

Em algoritmos de criptografia, a ideia de que o criptoanalista possui acesso apenas ao criptograma e, em alguns casos, ao texto claro, não pode ser considerada verdadeira quando o contexto é *white-box*. Em *white-box*, os algoritmos são executados no ambiente do cliente, o que permite ao detentor do *host* de execução, um nível de acesso muito superior ao previsto por aqueles que projetaram as cifras candidatas ao concurso AES, por exemplo.

Alguns algoritmos, como o DES e AES foram implementados também para o ambiente *white-box*, de forma a torná-los mais seguros nesse novo contexto; contudo, nem todos os algoritmos propostos para *black-box* possuem uma implementação que seja mais resistente em um ambiente vulnerável. A fim de identificar as características presentes em uma cifra projetada para *white-box*, esta dissertação realiza uma análise das técnicas aplicadas as implementações *white-box* do AES, para que seja possível torná-las mais genéricas e construir um conjunto de técnicas recomendáveis para cifras simétricas implementadas nesse contexto. Para demonstrar a viabilidade dessas técnicas, esse trabalho gera ainda dois estudos de caso, baseados no Twofish: O WBTwofish e o WBTwofish+. O primeiro utiliza técnicas que apesar de mais simples, não alteram o criptograma do algoritmo original; além disso, se utilizado em conjunto com as codificações e bijeções, essa implementação pode alcançar uma resistência de 2^{40} passos, necessários para que a chave seja inferida pela criptoanálise BGE-Attack, em cada uma das rodadas. Já o WBTwofish+, sob as mesmas condições descritas para o WBTwofish e, através de algumas técnicas propostas para o WBAES+, alcança em cada rodada, uma resistência contra o BGE-Attack de: 2^{104} passos em sua *sbox* e 2^{128} passos em todo algoritmo. No entanto, os criptogramas do WBTwofish+ e Twofish são distintos.

ABSTRACT

On cryptography algorithms, the idea that the cryptoanalyst has only access to the cryptogram and, in some cases, to the plain text, cannot be considered true when the context is white-box. Under the white-box context, the algorithms run on the client environment, which allows a superior access by the execution host owner than that which was previously predicted from the AES context algorithms designers, for example.

Some algorithms, like the DES and AES, were also implemented for white-box, in order to make them stronger in this new context; however, not all algorithms proposed for black-box have a more secure implementation for a vulnerable environment. In order to identify the characteristics present in a cipher designed for white-box, this thesis performs an analysis of the techniques applied to the white-box AES, so that it is possible to make them more generic and construct a set of recommended techniques for symmetric ciphers, implemented under this context. To show the feasibility of these techniques, this work further generates two case studies, based on Twofish: WBTwofish and WBTwofish+. The former uses techniques which, although simpler, do not change the original algorithm cryptogram; futhermore, if it is used with encodings and mixing bijections, this implementation can reach a resistance of 2^{40} steps against the BGE-Attack cryptoanalysis, in each one of the rounds. On the other hand, WBTwofish+, under the same conditions as WBTwofish, and through some techniques proposed for WBAES+, reach in each one of the rounds, a resistance against the BGE-Attack of: 2^{104} in its sbox and 2^{128} in the whole algorithm. However, the cryptograms of WBTwofish+ and Twofish are different.

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Sendo a informação tema central deste trabalho e a fim de que sua importância seja imediatamente apresentada; introduz-se aqui um dos primeiros trechos de "O livro dos códigos", por Singh (2011):

> *"Durante milhares de anos, reis, rainhas e generais dependeram de comunicações eficientes de modo a governar seus países e comandar seus exércitos. Ao mesmo tempo, todos estavam cientes das consequências de suas mensagens caírem em mãos erradas, revelando segredos preciosos a nações rivais ou divulgando informações vitais para forças inimigas. Foi a ameaçada interceptação pelo inimigo que motivou o desenvolvimento de códigos e cifras, técnicas para mascarar uma mensagem de modo que só o destinatário possa ler seu conteúdo."*

Através do descrito acima, é possível perceber que a concepção das cifras não é algo recente; no entanto, o desenvolvimento das cifras evoluiu ao longo dos anos, e as mensagens a serem protegidas, se traduzem hoje, principalmente, nas informações que representam valor para as organizações, nações e indivíduos.

Dentro do escopo da cifras, esta dissertação aborda uma classe denominada simétrica, em que a chave utilizada para cifrar/decifrar a mensagem é a mesma.

No ano de 2001 foi estabelecido pelo NIST(National Institute of Standards and Technology), um novo algoritmo padrão de encriptação (STANDARD, 2001), denominado AES(Advanced Encryption Standard), que tinha por objetivo substituir o DES(Data Encryption Standard)(KOU, 1997), que não apresentava mais as características de segurança até então aceitáveis.

Para o concurso AES, foram selecionados 5 algoritmos finalistas; sendo o Rijndael, o algoritmo vencedor. Entretanto, tanto esse algoritmo, como os demais finalistas, apesar de atenderem aos requisitos do concurso, não previam as características necessárias para um ambiente mais vulnerável; como pode ser constatado através dos trabalhos de Barak et al. (2001) e Chow et al. (2003), posteriores a definição do algoritmo AES.

As vulnerabilidades não previstas para o algoritmo vencedor do concurso AES, se tornaram expostas através de um novo ambiente de execução, denominado *white-box*. Dentre as vulnerabilidades apresentadas nesse ambiente, esta dissertação tratará especificamente da extração da chave criptográfica do algoritmo, realizada através da criptoanálise BGE-Attack, proposta por Billet et al. (2005).

O contexto *white-box* consiste basicamente na execução do algoritmo de criptografia na máquina do usuário ou cliente; ao invés de fazê-la em um servidor em que esse cliente não possui acesso, o que é a característica do ambiente *black-box*. A figura 1.1 apresenta um comparativo entre os recursos aos quais um possível atacante ou criptoanalista possui em um contexto *white-box* e em um *black-box*.

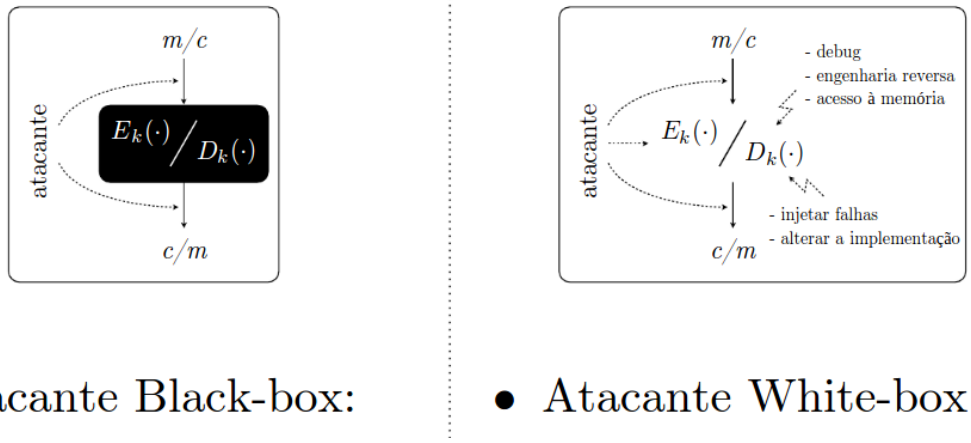

FIG. 1.1: Demonstração dos privilégios de um atacante em um ambiente *white-box*, em contrapartida aos recursos oferecidos em um ambiente *black-box*. Enquanto no ambiente *black-box*, o atacante possui acesso apenas ao texto de entrada e de saída do algoritmo, sem visibilidade da execução, no ambiente *white-box*, esse mesmo atacante acessa toda a implementação do software e possui controle sobre o ambiente de execução. Ilustração modificada de Lepoint et al. (2013)

Ao criar o ambiente *white-box*, Chow et al. (2003) propuseram duas implementações para esse novo contexto, sendo uma do algoritmo vencedor do concurso AES e a outra do DES. Essas implementações foram chamadas de WBAES e WBDES, respectivamente; e faziam uso de técnicas, como as que serão exemplificadas no capítulo 2 desta dissertação, para tornar mais difícil o acesso de um atacante a chave do algoritmo, sem alterar o criptograma da implementação original e assim permitir uma compatibilidade entre a implementação *white-box* e *black-box*. Durante toda a pesquisa, este trabalho abordará apenas a implementação *white-box* do AES, tendo em vista que o DES se mostrou vulnerável mesmo em um ambiente *black-box*.

Apesar das técnicas propostas por Chow et al. (2003) terem tornado o WBAES mais

resistente ao novo contexto, houve também a proposta de técnicas para extrair a chave criptográfica dessa implementação; como é o caso do BGE-Attack, proposto por Billet et al. (2005); que é capaz de extrair a chave do WBAES em 2^{24} passos, em cada rodada. Isso fez com que a implementação do WBAES continuasse a ser aprimorada, até resultar na implementação conhecida como WBAES+, que apesar de alterar o criptograma da implementação original do AES, elevou a resistência deste ao BGE-Attack para 2^{128} passos.

A fim de permitir que a evolução do AES em *white-box* possa ser estendida para outros algoritmos, de forma que estes apresentem maior resistência a ataques que visam a extração da chave criptográfica, como é o caso do BGE-Attack; esta dissertação analisa as características presentes tanto no WBAES, quanto no WBAES+; detalhando-as em capítulos e generalizando suas aplicações para gerar um conjunto de técnicas recomendáveis para algoritmos simétricos em um contexto *white-box*, sendo essa a motivação principal desta dissertação.

Além disso, para demonstrar a aderência das técnicas tratadas, serão apresentados dois estudos de caso: O WBTwofish e o WBTwofish+, ambos construídos a partir do Twofish, um dos finalistas do concurso AES. O primeiro estudo de caso, WBTwofish, faz uso principalmente das técnicas abordadas pelo WBAES, o que permite que o criptograma originalmente gerado pelo Twofish seja preservado. Já o WBTwofish+, utiliza também algumas funções projetadas pelo WBAES+; o que faz com o que criptograma gerado seja distinto do produzido pelo Twofish.

1.2 OS AMBIENTES DE ACESSO

1.2.1 O AMBIENTE *BLACK-BOX*

É o ambiente em que os algoritmos de criptografia são convencionalmente executados, não havendo visibilidade dos dados persistidos em cada etapa de execução do algoritmo. A informação acessível ao criptoanalista se limita ao criptograma e, em alguns casos, o texto claro; quando se trata de um ataque de texto escolhido, por exemplo.

1.2.2 O AMBIENTE *GRAY-BOX*

Para tratar sobre o ambiente *gray-box*, Chow et al. (2003), que são pioneiros da implementação *white-box*, iniciam definindo os verdadeiros ataques *black-box*, que em sua concepção não necessitam de qualquer conhecimento sobre o funcionamento do algoritmo, sendo compostos por ataques genéricos. Já os ataques mais avançados, que exploram detalhes do

funcionamento do algoritmo que se deseja atacar, são categorizados por eles como sendo do tipo *gray-box*.

Nessa categoria de ataques, tem-se aqueles que incluem a criptoanálise linear e diferencial, além dos conhecidos como *side-channel* ou *partial-access*, que incluem análise de tempo, energia e falhas, durante a execução do programa.

1.2.3 O AMBIENTE *WHITE-BOX*

Além do contexto em que ocorrem os ataques *black-box* e *gray-box*, existe também o ambiente denominado *white-box*, em que a situação é mais severa, tendo em vista que um usuário, detentor do *host* de execução e com acesso à implementação do algoritmo, pode ser classificado como um possível atacante. Exemplos de contexto *white-box* podem ser encontrados em:

- Aplicações executadas em *tablets*, *smartphones* ou em qualquer *host* de execução que esteja em poder do usuário final.
- Alguma aplicação na nuvem em que a execução ocorra no *client-side*.
- Softwares distribuídos para usuários finais como parte de algum sistema de DRM (*Digital Rights Management*).
- Algoritmos de criptografia executados em *smart-cards*.

Os ataques normalmente aplicados aos ambientes *black-box* ou *gray-box*, como os já citados *side-channel* ou *partial-access* serão desconsiderados, pois o acesso a implementação se torna muito mais interessante do que informações secundárias obtidas nesses ambientes.

Supervisionar a variação dos recursos consumidos por um algoritmo durante a sua execução pode apontar informações pertinentes para o seu funcionamento, no entanto, as informações fornecidas em um ambiente *white-box* são muito mais relevantes para o atacante que deseja extrair a chave criptográfica. Dentre os recursos disponíveis nesse contexto, pode-se citar, por exemplo:

- Acesso completo e privilegiado ao *host* em que o software está sendo executado e à implementação do algoritmo, como ocorre em *scripts*, por exemplo, em que a execução do código pode ocorrer no *client-side*.

- Detalhes internos do algoritmo que podem ser observados e alterados durante o processo de encriptação.

É possível perceber mais claramente as diferenças existentes entre os ambientes *white-box* e *black-box*, através da figura 1.1; que apresenta os recursos que o atacante tem acesso em cada um dos contextos.

1.3 OBJETIVOS DE PESQUISA

Dentre os objetivos desta dissertação, pode-se listar os seguintes:

- Apresentar de forma detalhada as técnicas que permitiram ao WBAES e ao WBAES+ aprimorar a segurança do AES em um contexto *white-box*.
- Aplicar algumas das técnicas implementadas pelo WBAES e WBAES+ em outro algoritmo sem ser o AES e analisar os resultados obtidos por essa aplicação.

1.4 PROBLEMAS DE PESQUISA

A partir dos objetivos definidos na seção anterior, tem-se que algumas questões são levantadas:

- Pode-se gerar um conjunto de técnicas recomendáveis para algoritmos simétricos em um contexto *white-box*, a partir de uma análise do WBAES e do WBAES+?
- Os resultados obtidos pela aplicação em outros algoritmos das técnicas utilizadas pelo WBAES e WBAES+, se aproximam do resultado apresentado pelo AES e derivados?

1.5 ORGANIZAÇÃO DO TRABALHO

Esta dissertação foi estruturada de forma a permitir que os elementos mais relevantes na implementação *white-box* fossem separados por capítulos e tratados de maneira quase que cronológica à sua proposição.

O segundo capítulo apresenta as tabelas de pesquisa propostas por Chow et al. (2003) e demonstra a importância de se reestruturar as funções do algoritmo para minimizar o vazamento de informações que levem à chave criptográfica.

O terceiro capítulo aborda a confusão, exemplificando a sua importância para os algoritmos de criptografia. Além disso, são tratados recursos que permite o aprimoramento da confusão, como as codificações de Chow et al. (2003), por exemplo.

No quarto capítulo são abordadas técnicas que permitem aprimorar a difusão do algoritmo e, é demonstrado o uso de matrizes MDS maiores para se alcançar o efeito avalanche em um único *round*; além disso, são apresentadas as bijeções misturadoras de Chow et al. (2003), que também aprimoraram a difusão do algoritmo.

Já o capítulo 5, aborda as funções do algoritmo WBAES+ que apresentam dependência de chave e demonstram quais procedimentos devem ser realizados para fazer com que funções que atuavam com valores estáticos, passem a fazer uso de valores dinâmicos de acordo com a chave.

O capítulo 6 reúne algumas práticas para aqueles que desejam portar suas implementações para o ambiente *white-box* e demonstra a aplicação de algumas técnicas em dois estudos de caso: O WBTwofish, que não altera o criptograma do Twofish e, o WBTwofish+, que estende o funcionamento do WBTwofish, rompendo a compatibilidade com o criptograma do Twofish, a fim de tornar o algoritmo mais resistente em um contexto *white-box*.

Por fim, o capítulo 7 apresenta uma análise final do trabalho, com uma breve listagem das contribuições fornecidas, seguida por algumas sugestões de trabalhos futuros.

2 TABELAS DE PESQUISA

Ao analisar o AES e DES, Chow et al. (2003) verificaram que a estrutura desses algoritmos armazenava informações na memória que permitiam a dedução da chave criptográfica de forma simples. Essa dedução ocorria através do resgate do valor armazenado na memória, que havia sofrido poucas transformações; essas transformações eram revertidas, devido ao conhecimento público da estrutura do algoritmo, e a chave obtida.

A fim de minimizar isto, Chow et al. (2003) adotaram para a sua estrutura *white-box* uma implementação do AES em que as funções eram reorganizadas e aglutinadas antes que seu resultado fosse armazenado na memória.

No contexto *white-box*, o atacante tem acesso direto a memória, logo, armazenar o resultado de operações diversas, envolvendo funções do algoritmo de criptografia ou outras para adicionar confusão e difusão junto à chave, ao invés de apenas esta diretamente, permite introduzir um maior esforço para que o atacante consiga extrair essa chave do algoritmo; o que se traduz em um aumento na complexidade de execução da técnica empregada pelo criptoanalista, como por exemplo, se comparados os custos de execução do BGE-Attack sobre o WBAES e WBAES+.

Essas informações aglutinadas são as saídas das operações reestruturadas no algoritmo; denominadas por Chow et al. (2003) como tabelas de pesquisa ou *lookup-tables*. Elas consistem basicamente em um mapeador de *bits*, que contém m *bits* em sua entrada e n *bits* em sua saída. Apesar de refletir um funcionamento semelhante ao realizado pelas funções convencionais do algoritmo de criptografia, essa estrutura permite uma melhor representação gráfica e um maior acoplamento das antigas funções, agora tabelas de pesquisa, no processo de transformação do texto claro em criptograma.

Em um ambiente *white-box*, para se obter uma tabela de pesquisa que possua a mesma segurança apresentada em um contexto *black-box*, uma das possibilidades é criar uma tabela de pesquisa que para cada entrada possível, forneça 2^{Tb} saídas correspondentes. Sendo Tb o tamanho do bloco, que normalmente é igual a 64 ou 128, essa tabela de pesquisa hipotética deve fornecer para cada um dos 16 *bytes* do *state*, considerando que o tamanho do bloco é igual a 128, 2^{128} *bits* correspondentes; que na notação utilizada por Chow et al. (2003) é representado por ($2^{Tb} \cdot 16$ *bytes*). No entanto, a aplicação desta tabela de pesquisa é inviável, devido ao seu custo computacional.

Pela impossibilidade de construção de uma tabela que contemplasse todas as possi-

bilidades de mapeamento de *bits* em função do tamanho do bloco; Chow et al. (2003) construíram para a sua implementação *white-box*, com base no AES, algumas tabelas de pesquisa menores. Algumas dessas tabelas realizavam os mesmos mapeamentos executados pelas funções do AES, enquanto outras aprimoravam a confusão e difusão do algoritmo; o que tornava a implementação de Chow et al. (2003) mais eficiente em um ambiente *white-box* do que a convencional.

Esse capítulo tratará apenas das tabelas construídas a partir das funções do AES. Já os capítulos seguintes, apresentarão as tentativas de generalização das estruturas aplicadas ao WBAES e WBAES+, a fim de permitir a aplicação dessas estruturas em outros algoritmos no ambiente *white-box*; além de tratarem sobre outros conceitos evoluídos ao longo da implementação *white-box* para aumentar a segurança da chave.

A seção seguinte apresentará de forma concisa o funcionamento de cada uma das funções do AES, para que seja possível compreender exatamente as operações que serão executadas pelas tabelas de pesquisa criadas através destas.

2.1 AS FUNÇÕES DO AES

2.1.1 *SHIFTROWS*

Na transformação *ShiftRows*, os *bytes* das últimas três linhas do *state* são permutados de forma cíclica, de acordo com o índice de sua linha. Para a primeira linha, em que o índice r é igual a 0, não há permutação.

O *ShiftRows* considera além da variável r: Nb[1], c[2] e a função *shift(r, Nb)*, que marca os deslocamentos realizados pelo *Shiftrows* em cada linha.

Como o número de colunas da matriz não é alterado, uma vez que este é definido, Nb manterá o seu valor igual a 4 até o fim da execução do algoritmo, enquanto r apresentará valores que oscilam entre 1 e 3, conforme apresenta a figura 2.1, que define tanto o valor de r quanto de c.

A permutação realizada pelo *ShiftRows* move os *bytes* com índices c maiores para posições menores, enquanto aqueles com um menor índice c serão deslocados para posições maiores. A quantidade de deslocamentos que um determinado *byte* sofre, é determinada em função do índice de sua linha, r. Esse comportamento é ilustrado na figura 2.2, que demonstra o *ShiftRows* em execução.

[1]Representa o número de colunas da matriz
[2]Indica o índice de cada coluna

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \bmod Nb} \quad \text{para } 0 < r < 4 \text{ e } 0 \leq c < \mathbf{Nb}$$

$$shift(1,4) = 1\,; \quad shift(2,4) = 2\,; \quad shift(3,4) = 3\,.$$

FIG. 2.1: Representação da função *ShiftRows*, permutando os *bytes* de acordo com os índices r e c, que indicam respectivamente linha e coluna da matriz a ser alterada. Ilustração extraída de Standard (2001)

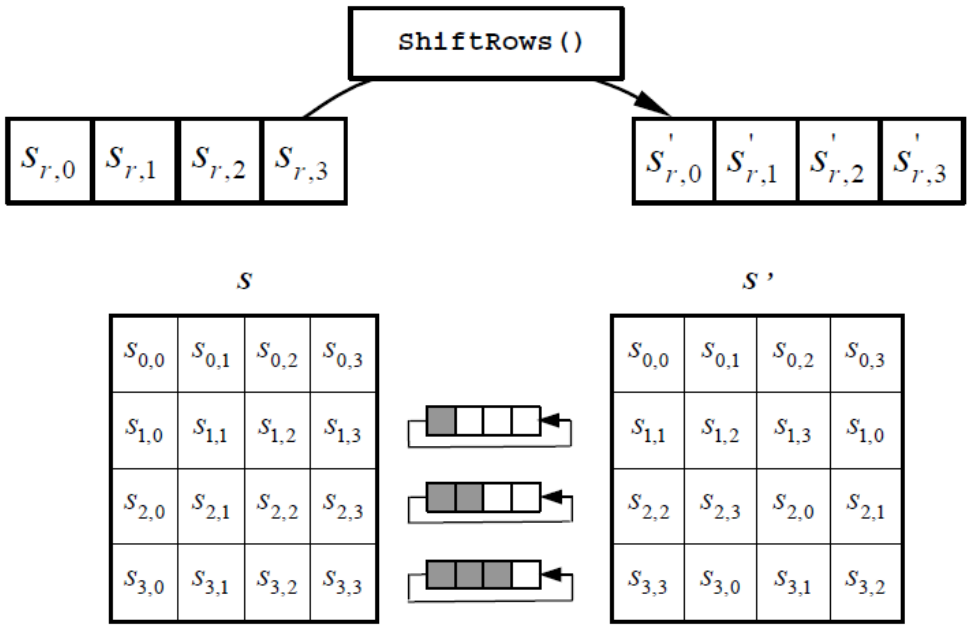

FIG. 2.2: ShiftRows, deslocando as três últimas linhas do State; extraída de Standard (2001).

2.1.2 *SUBBYTES*

O *SubBytes* é uma função não-linear, que opera em cada *bytes* do *state* de forma independente. A S-box utilizada na transformação *SubBytes* é apresentada na base hexadecimal na figura 2.3. Essa substituição é determinada utilizando-se os valores do *byte* a ser substituído como coordenadas na matriz de substituição. Por exemplo, para $S_{1,1} = \{53\}$, o primeiro valor desse *byte*, '5', corresponderá a quinta linha na matriz de substituição, enquanto '3', será equivalente a terceira coluna. Através da interseção desses pontos, tem-se como resultado $S'_{1,1} = \{\text{ed}\}$.

2.1.3 *MIXCOLUMNS*

O *MixColumns* combina cada coluna do *state* por vez, utilizando uma transformação linear inversa. A função *MixColumns* recebe quatro *bytes* como entrada e retorna quatro

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

FIG. 2.3: *Sbox*: Matriz de substituição com valores em hexadecimal; extraída de Standard (2001).

em sua saída; cada um desses *bytes* de entrada afeta todos os quatro *bytes* de saída.

Em conjunto com o *ShiftRows*, o *MixColumns* adiciona difusão a cifra; realizando a multiplicação de cada coluna do *state* por uma matriz estática, contendo 128 *bits*. Como demonstrado na figura 2.4.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \qquad \text{for } 0 \le c < \boldsymbol{Nb}.$$

FIG. 2.4: Multiplicação realizada entre os *bytes* de uma determinada coluna do *state* e da matriz de 128 *bits* do *MixColumns*. Ilustração extraída de Standard (2001)

2.1.4 *ADDROUNDKEYS*

O *AddRoundKey* é uma função que realiza um XOR entre duas matrizes, sendo aplicado a cada um dos *bytes* das duas matrizes que apresentem o mesmo índice de linha e coluna e ao final retornando uma terceira matriz como resultado da combinação dessas duas.

2.1.5 EXECUÇÃO CONVENCIONAL DO AES

Após a descrição de suas funções, o algoritmo 1, que representa a execução convencional do AES-128, é apresentado sem qualquer tipo de alteração em sua estrutura. Nas próximas seções, será apresentado o processo evolutivo para se transformar esse algoritmo no modelo proposto por Chow et al. (2003).

Algoritmo 1: IMPLEMENTAÇÃO CONVENCIONAL AES-128

```
1  início
2      state = plaintext
3      AddRoundKey(state, k_0)
4      para r = 1 até 9 faça
5          SubBytes(state)
6          ShiftRows(state)
7          MixColumns(state)
8          AddRoundKey(state, k_r)
9      fim
10     SubBytes(state)
11     ShiftRows(state)
12     AddRoundKey(state, k_10)
13     ciphertext = state
14 fim
```

2.2 REORGANIZANDO AS FUNÇÕES DO AES PARA A CRIAÇÃO DAS TABELAS

Para não armazenar a chave diretamente na memória e visando uma maior aglutinação das operações executadas pelo novo AES, adaptado para o contexto *white-box*, Chow et al. (2003) resolveram descrever tanto as funções já existentes no AES convencional, quanto aquelas adicionadas por eles para aumentar a confusão e difusão do algoritmo, em formato de tabelas de pesquisa.

Visando utilizar esse novo formato, Chow et al. (2003) adotaram uma implementação alternativa para o AES, em que a chave criptográfica fosse operada de modo mais interno no algoritmo, alterando a ordem de execução das funções da estrutura original da implementação.

A reorganização dessas funções, embora simples, permitiu que antes mesmo de sua transformação em tabelas de pesquisa, fosse possível armazenar na memória, uma infor-

mação com mais alterações do que apenas um XOR realizado entre a chave e o texto claro, fornecido pelo *AddRoundKey*; conforme é apresentado nos algoritmos 1 e 2.

Algoritmo 2: IMPLEMENTAÇÃO ALTERNATIVA AES-128

```
1  início
2      state = plaintext
3      para r = 1 até 9 faça
4          ShiftRows(state)
5          AddRoundKey(state, ShiftRows(k_{r-1}))
6          SubBytes(state)
7          MixColumns(state)
8      fim
9      ShiftRows(state)
10     AddRoundKey(state, ShiftRows(k_9))
11     SubBytes(state)
12     ShiftRows(state)
13     AddRoundKey(state, ShiftRows(k_10))
14     ciphertext = state
15 fim
```

2.3 ANALISANDO A ESTRUTURA DO ALGORITMO PARA ADAPTÁ-LO AO AMBIENTE *WHITE-BOX*

Analisar a estrutura do algoritmo executado em um ambiente *black-box*, é um fator importante antes de portá-lo para o ambiente *white-box*. A reorganização das funções, apresentada no algoritmo 2, pode ser desmembrada em conceitos que se aplicam a qualquer algoritmo.

Através da comparação da implementação convencional do AES, apresentada no algoritmo 1, e da implementação alternativa, adotada por Chow et al. (2003), apresentada pelo algoritmo 2, pode-se pontuar suas diferenças, mesmo em uma análise que contemple apenas as linhas iniciais do código. Por exemplo, da análise da terceira linha do algoritmo 1, repetida abaixo,

$$AddRoundKey(state, k_0)$$

Percebe-se que a ocorrência da função *AddRoundKey* sobre o *state* e o k_0, que é

a chave principal usada na implementação do algoritmo, implica no armazenamento de uma informação na memória que permite resgatar k_0, realizando-se apenas um outro *AddRoundKey* sobre o resultado anterior e o *state*; o que retornará a chave. Portanto, tem-se que:

$$x = AddRoundKey(state,\ k_0)$$
$$k_0 = AddRoundKey(state,\ x)$$

Logo, em apenas um passo após a ocorrência da função $AddRoundKey(state,\ k_0)$; é determinado por um atacante o valor de k_0. Já no algoritmo 2, é apresentada a implementação alternativa adotada por Chow et al. (2003); contudo, uma pequena alteração na ordem de ocorrência das funções *ShiftRows*, *AddRoundKey* e *SubBytes*, impacta em um aumento no número de passos entre o atacante e a chave, k_{r-1}.

$$ShiftRows(state)$$
$$AddRoundKey(state,\ ShiftRows(k_{r-1}))$$

Logo, para se recuperar k_{r-1} e, considerando a nova função *inv_ShiftRows* como a inversa da função *ShiftRows*, tem-se que:

$$ShiftRows(state)$$
$$x = AddRoundKey(state,\ ShiftRows(k_{r-1}))$$
$$y = AddRoundKey(state,\ x)$$
$$k_{r-1} = inv\_ShiftRows(y)$$

O *ShiftRows* executa uma permutação nos *bytes* do *state* que, segue como critério de deslocamento, o índice de cada linha ou *row* do *state*; sendo o número de *bytes* deslocados igual a 0 na linha 0, 1 na linha 1 e assim sucessivamente.

Essa função também é aplicada sobre a chave, k_{r-1}. Sendo Δ, o número de casas deslocadas pelo *ShiftRows* em cada linha, esse deslocamento será o mesmo para ambas as matrizes, *state* e k_{r-1}; independente do número de vezes que essa função seja executada sobre essas matrizes.

Como o deslocamento é igual para as duas matrizes, *state* e k_{r-1} e, sendo o *AddRoundKey* aplicado sobre os *bytes* das duas matrizes individualmente, logo, o XOR será realizado entre os mesmos *bytes* que o fariam antes da ocorrência do *ShiftRows*.

Apesar de ainda haver pequenas mudanças entre a implementação do algoritmo 2 e a estrutura convencional do AES, já é possível perceber que se faz necessário desfazer duas funções, *AddRoundKey* e *ShiftRows* para se obter a chave, k_{r-1}. Essa implementação

ainda sofrerá diversas modificações até atingir as tabelas de pesquisa propostas por Chow et al. (2003); contudo, mesmo após essas modificações, o criptograma utilizando tabelas de pesquisa[3] resultará no mesmo da implementação convencional do AES, se utilizado o mesmo texto claro, chave e especificações de cifragem. Isso é demonstrado por uma implementação realizada em C++, fornecida por esta dissertação; que demonstra as funções do AES escritas da forma que o WBAES propõe.

Assim como o *AddRoundKey*, que se aplica a apenas um *byte* por vez, outras operações que atuem individualmente sobre os *bytes* podem ser precedidas por funções de permutação, como o *ShiftRows*, sem que haja mudança nos *bytes* afetados. Caso a função, relacione os *byte* de duas matrizes, como é feito pelo *AddRoundKey*, é necessário que se mantenha o mesmo deslocamento dos *bytes*, conforme pontuado anteriormente.

2.4 REMODELANDO AS FUNÇÕES EM TABELAS DE PESQUISA

Apesar de a descrição alternativa do AES, demonstrada na seção anterior, elevar o número de operações necessárias para o atacante extrair a chave criptográfica, ainda é necessário explorar a estrutura apresentada no algoritmo 2; de modo a realizar o máximo de aglutinações possíveis, gerando tabelas de pesquisa que elevem ainda mais o número de passos a serem executados por um determinado atacante para que este possa resgatar a chave do algoritmo.

Nas subseções seguintes, serão descritas apenas as tabelas de pesquisa construídas a partir das funções do AES. Para exemplificar o funcionamento dessa nova estrutura, a figura 2.5 demonstra qual o esquema será alcançado ao combinar as *TBoxes*, representadas no diagrama por T; as Ty_i *Tables*, representadas por Ty_i; e as *XORTables*, que são resumidas apenas em XOR. Cada uma dessas tabelas de pesquisa será detalhada a seguir, demonstrando-se o processo realizado para a representação do AES em um contexto *white-box*.

2.4.1 *TBOXES*

Contemplam as primeiras operações executadas pelo AES; sendo construídas a partir da percepção de que aglutinar as operações na memória, ao invés de persistir o resultado dessas funções de forma individual, torna mais custoso para o atacante distinguir a chave criptográfica. Na implementação WBAES, as *TBoxes* surgiram como uma estrutura que

[3]Considerando apenas a modificação estrutural nas tabelas, não foi testado se o criptograma se mantinha inalterado em relação ao original com a utilização de técnicas para aprimorar a confusão, difusão e introduzir ofuscação.

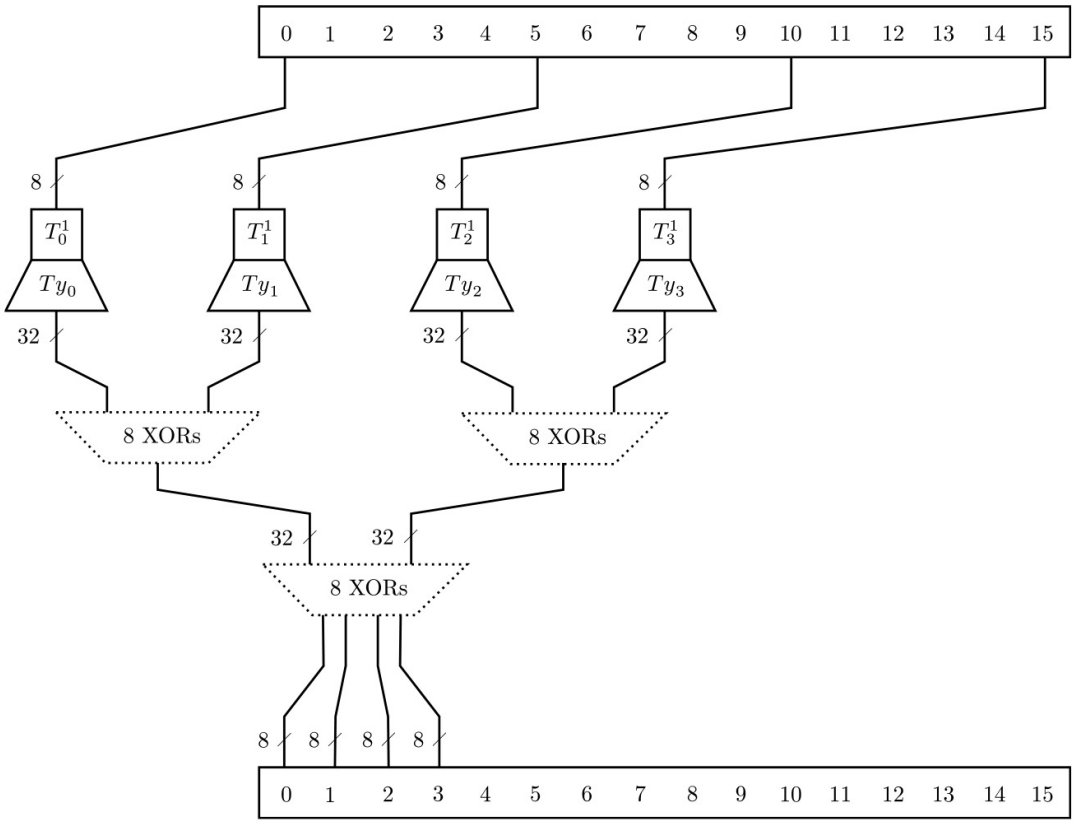

FIG. 2.5: Representação de um *round* do AES-128 em formato de tabela de pesquisa, *round* 1. Funcionamento ilustrado por Muir (2013).

reunia as funções *SubBytes* e o *AddRoundKey*; que se tornaram imediatamente conectadas pela nova estrutura do AES, conforme demonstra o algoritmo 2.

A entrada das *TBoxes* é construída a partir das matrizes *state* e *k*, que em cada round possuem um índice para representar o *round key*; antes de entrarem nas *TBoxes*, os *bytes* dessas matrizes são permutados pela função *ShiftRows*[4], que mantém o mesmo Δ para as matrizes. Após serem introduzidos nas *TBoxes*, esses *bytes* são combinados através da operação XOR, realizada pela função *AddRoundKey* e substituídos pela função *SubBytes*.

A seguir é representada a estrutura das *TBoxes* para todos os *rounds* do AES. A função T corresponde as *TBoxes*; *x[i]* representa o *byte* do *state* e *k[i]* o *byte* da chave, que terão o seu índice determinado de acordo com a chave do *round*. Os *bytes* de k que forem representados por $\hat{k}$, correspondem a chave do *round* que foi permutada pelo *ShiftRows*. A única exceção é a chave k_{10}, conforme mostra o algoritmo 2.

T_i^r = SubBytes(AddRoundKey(x[i], $\hat{k}_{r-1}$[i])), para i = 0...15 e r = 1...9,

T_i^{10} = AddRoundKey(SubBytes(AddRoundKey(x[i], $\hat{k}_9$[i])), k_{10}), para i = 0...15

[4]Exceto os *bytes* da matriz k_{10}

2.4.2 *TYITABLES*

Logo após a ocorrência das *TBoxes* (*AddRoundKey* + *SubBytes*), são executadas as $Ty_i\,Tables$(*MixColumns*), que em sua entrada recebem os *bytes* de saída das *TBoxes*. As $Ty_i\,Tables$ representam o funcionamento do *MixColumns*, em formato de tabela de pesquisa; sendo aplicadas entre os rounds 1 e 9, assim como o *MixColumns*.

Tomando como exemplo as *TBoxes* do *round* 1: T_0^1, T_1^1, T_2^1, T_3^1; tem-se que a saída dessas *TBoxes* podem ser interpretadas como um vetor de coluna; já que juntas, estas podem constituir uma matriz unidimensional. Em seguida, esse vetor será multiplicado pela matriz *MC*.

Assim como na subseção que tratou sobre as *TBoxes*, x será utilizado para representar os *bytes* do *state* que serão multiplicados pela matriz *MC*; por serem quatro *bytes*, os índices de x variam entre 0 e 3; sendo x_0, x_1, x_2, x_3, os valores representados. A multiplicação desses *bytes* é demonstrada na figura 2.6.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_0 \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus x_1 \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus x_2 \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus x_3 \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}$$

FIG. 2.6: Multiplicação de *bytes* do *state* pela matriz *MC*. Ilustrado por Muir (2012)

Cada um desses *bytes* será a entrada das tabelas 8x32 *bits*, denominadas $Ty_i\,Tables$; sendo cada uma dessas tabelas limitadas a 256 valores possíveis, em função de sua entrada, conforme apresenta Muir (2013) na figura 2.7.

Esse limite de possibilidades faz com que seja necessário o uso de segurança local, como será explicado nas próximas seções, para inviabilizar que os valores dessas tabelas sejam inferidos individualmente.

As $Ty_i\,Tables$ realizam quatro mapeamentos de 8x32 *bits* e, que deverão resultar em um único mapeamento de 32x32 *bits*, após a realização das operações XOR necessárias, que foram isoladas em uma tabela de pesquisa, conforme será demonstrado na subseção seguinte. Apesar de realizar de forma simplificada o apresentado a seguir para efetuar a junção das $Ty_i\,Tables$, uma melhor representação desse processo é apresentada na figura 2.8, que consiste em uma especificação da conexão entre as tabelas *TBoxes*, T; $Ty_i\,Tables$, Ty_i e *XORTables*, XOR. Já apresentada na figura 2.5, no início desta seção.

$$\mathrm{T}y_0(x_0) \oplus \mathrm{T}y_1(x_1) \oplus \mathrm{T}y_2(x_2) \oplus \mathrm{T}y_3(x_3).$$

$$
\begin{aligned}
Ty_0(x) &= x \cdot [02\ 01\ 01\ 03]^{\mathsf{T}} \\
Ty_1(x) &= x \cdot [03\ 02\ 01\ 01]^{\mathsf{T}} \\
Ty_2(x) &= x \cdot [01\ 03\ 02\ 01]^{\mathsf{T}} \\
Ty_3(x) &= x \cdot [01\ 01\ 03\ 02]^{\mathsf{T}}.
\end{aligned}
$$

FIG. 2.7: Representação da multiplicação dos *bytes* do *state* pela matriz *MC* em vetor de coluna; resultando nas Ty_i*Tables* correspondentes a cada índice.

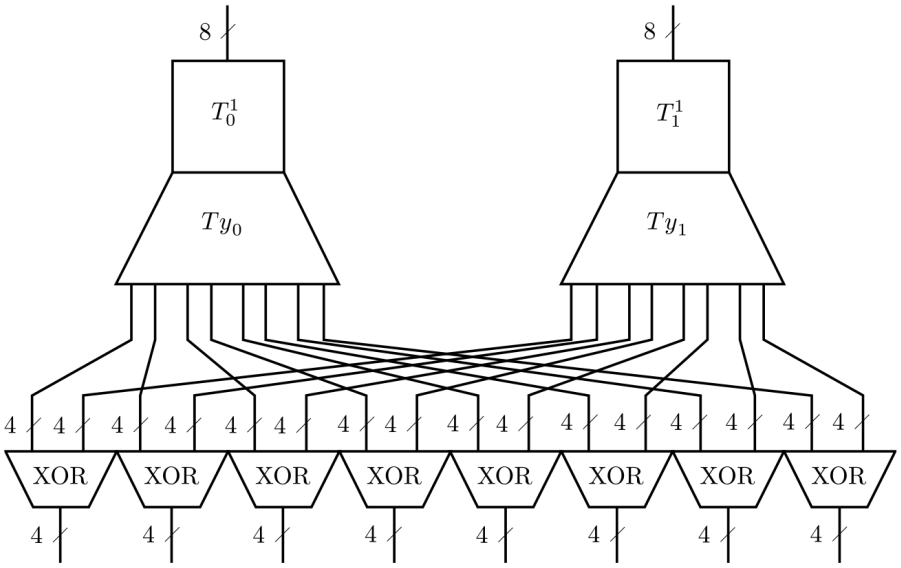

FIG. 2.8: Ou-exclusivo de duas tabelas Ty_i. Para realizar essa operação foram necessárias 8 tabelas XOR, que combinaram os 64 *bits* de saída de duas Ty_i*Tables*.

2.4.3 *XORTABLES*

São tabelas responsáveis por realizar a operação de ou-exclusivo, ou XOR, entre dois *nibbles*(4 *bits*). Essas tabelas de pesquisa possibilitam a realização das Ty_i*Tables* e das *TBoxes*Ty_i*Tables*, que utilizam o XOR entre os *nibbles* de seus *bytes* para combinarem seus *bits* de saída.

Além disso, o fato de possuírem dois *nibbles* como entrada, realizando um mapeamento 8x4, ao invés de 16x8, realizado pelo *AddRoundKey*, permite realizar mais do que apenas duas combinações entre os *bytes* de tabelas de pesquisa distintas. Como foi mostrado na figura 2.8, cada tabela de pesquisa fornece metade do *byte* para a realização do XOR.

2.4.4 *TBOXESTYITABLES*

São tabelas geradas a partir da aglutinação das *TBoxes* com as Ty_i *Tables*, tendo em vista que entre os rounds 1 e 9, as *TBoxes* são entradas diretas das Ty_i *Tables*, podendo-se criar uma só tabela.

Para o *round* 1, por exemplo, as tabelas T_0^1 e Ty_0 podem ser substituídas por uma $Ty_0(T_0^1(x))$, reduzindo-se o número de acessos individuais e persistindo uma informação ainda mais condensada na memória. A ocorrência de Ty_i *Tables* após as *TBoxes*, implicará na criação de *TBoxes*Ty_i *Tables*, sempre que essa aglutinação fizer sentido.

2.4.5 SOBRE AGLUTINAR AS *TBOXES* COM AS *TYITABLES*

Para os demais algoritmos simétricos, será observado que fatalmente a saída de determinadas funções, será a entrada de outras e, sempre que houver a possibilidade de aglutinação, é importante que a considere. Analisando, por exemplo, em qual momento da execução do algoritmo essa aglutinação pode acontecer e se o benefício fornecido por esta é justificável no momento de sua aplicação.

Esse *time* para aplicação da aglutinação é importante, pois caso o algoritmo esteja em uma fase em que a possibilidade de se deduzir a chave criptográfica principal, a partir dessa tabela de pesquisa seja muito pequena, não se faz relevante aplicá-la, pois o ataque estará focado em uma fase que antecede a aglutinação.

2.4.6 RESULTADO

Embora esta seção tenha tratado apenas da transformação das funções do AES em tabelas de pesquisa, essa análise dos processos realizados por Chow et al. (2003), permite adaptar conceitos necessários para se transformar as funções de outros algoritmos simétricos do contexto *black-box* para *white-box*, reorganizando, aglutinando, generalizando ou mesmo alterando o funcionamento dessas funções, caso isto se faça necessário.

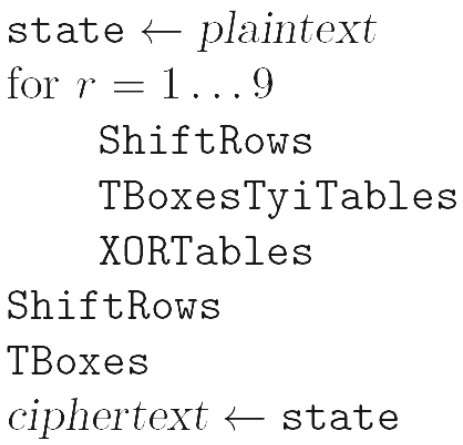

FIG. 2.9: Implementação do AES-128 em um contexto *white-box* após a aplicação das tabelas de pesquisa. Ilustrado por Muir (2013).

Até aqui, a implementação da figura 2.9, que representa as mudanças realizadas nas funções do AES para torná-las tabelas de pesquisa, resulta no mesmo criptograma do modelo do algoritmo 1. Conforme apresenta a implementação em C, produzida por esta dissertação.

Contudo, nos próximos capítulos, serão demonstradas outras técnicas para minimizar o vazamento da chave criptográfica em um ambiente *white-box* e nem sempre será possível preservar todas as características do algoritmo de criptografia original em função da segurança.

3 CONFUSÃO

Este capítulo trata sobre o conceito denominado confusão, que foi apresentado por Shannon (1949), em seu artigo: *"Communication Theory of Secrecy Systems"*. Nesse artigo, Shannon (1949) demonstrou que o emprego da confusão é essencial para projetos de algoritmos criptográficos, sendo uma das consequências de seu uso adequado, o aumento da resistência contra ataques estatísticos.

3.1 A CONFUSÃO SEGUNDO SHANNON

Shannon (1949) discorre sobre meios para se obter informações sobre a chave ou o texto claro, através de uma análise estatística realizada sobre o criptograma. O autor enumera os requisitos desejáveis para uma boa análise estatística e em seguida apresenta dois métodos que visam frustrar este tipo de análise, sendo estes chamados de confusão e difusão. O primeiro será tratado neste capítulo e contempla a relação existente entre a chave e o criptograma, enquanto o segundo método, difusão, será tratado no capítulo seguinte e corresponde ao relacionamento estabelecido entre o texto claro e o criptograma.

O objetivo da confusão é tornar a relação existente entre a chave criptográfica, K, e o texto cifrado ou criptograma, E, tão complexa quanto possível. No entanto, mesmo em uma relação mais complexa, ainda é permitido a um atacante estabelecer uma análise estatística entre K e E. Através dessa análise, pode-se limitar os possíveis *bytes* de K em função dos *bytes* de E. Esse conjunto de possibilidades para se determinar os *bytes* de K é representado abaixo pelos s_i*'s*, sendo cada s_i decorrente da aplicação de uma função f sobre os *bytes* de K.

$$
\begin{aligned}
f_1(k_1,\, k_2,\ldots,\, k_p) &= s_1 \\
f_2(k_1,\, k_2,\ldots,\, k_p) &= s_2 \\
&\ldots \\
f_n(k_1,\, k_2,\ldots,\, k_p) &= s_n,
\end{aligned}
$$

Em um cenário em que é introduzida a confusão, é necessário que o criptoanalista estabeleça a relação existente entre todos f_i's e seus respectivos k_i's, a fim de se obter todos os *bytes* de K. Nesse cenário, o criptoanalista deve solucionar todo o sistema de forma simultânea, já que não há a possibilidade de se obter os k_i's nas equações mais simples e

posteriormente substituí-los nas mais complexas. Como ocorreria em um contexto sem a confusão.

Para exemplificar esse funcionamento, pode-se comparar o comportamento de um sistema em que a confusão não é introduzida, com o de uma cifra de substituição monoalfabética, onde uma vez estabelecido k_2 para f_3, por exemplo, este também seria determinado para os demais f_i's. Já em um sistema em que é utilizada a confusão, existiria, de forma análoga, um alfabeto diferente para cada f_i, não sendo estabelecida conexão entre os *bytes* de K para f_i's distintos.

3.1.1 O MÉTODO DA PALAVRA PROVÁVEL

O método da palavra provável, como o próprio nome sugere, consiste em utilizar a probabilidade de ocorrência de palavras específicas na mensagem ou texto claro para efetuar uma análise estatística mais assertiva no criptograma. Essas palavras variam de acordo com o idioma e conforme Shannon (1949), as palavras ou sílabas *such*, *as*, *the*, *and*, *tion* e *that* apresentam grande chance de compor a mensagem; quando se trata de um texto em inglês, como era o cenário abordado pelo artigo e, portanto, ter a sua correspondência determinada no criptograma.

Esse método foi apresentado por Shannon (1949) como um artifício poderoso em uma análise estatística, este permite utilizar possíveis *bytes* de entrada para tornar mais efetiva uma análise estatística sobre os *bytes* de saída. Contudo, o método da confusão considera que mesmo que muitos *bytes* de entrada, texto claro, e *bytes* do criptograma sejam conhecidos, não seja uma tarefa simples conhecer o *bytes* da chave, K.

Conforme demonstra Shannon (1949) através de notação funcional, tanto as operações de cifragem

$$E = f(K,\, M),$$

quanto as de decifragem

$$M = g(K,\, E),$$

podem ser operações simples, sem que seja simples a operação que determina a chave:

$$K = h(M,\, E).$$

A confusão permite inserir algumas dificuldades na utilização da técnica das palavras prováveis por um criptoanalista. Em um contexto *white-box* supõe-se que o atacante conhece tanto o texto claro, M, quanto o criptograma, E, sendo a chave, K, o único elemento a ser desvendado por este.

3.2 A UTILIZAÇÃO DA CODIFICAÇÃO NO WBAES

Ao introduzir as limitações de segurança em um ambiente *white-box*, Chow et al. (2003)' apresentaram dois cenários em que a chave criptográfica poderia ser manipulada neste ambiente supervisionado sem que fosse diretamente exposta na memória e consequentemente resgatada por um atacante.

O primeiro cenário consistia em uma chave fixa, que estivesse dentro da própria implementação do algoritmo, de modo *hard-code*, por exemplo, ou que já tivesse sido introduzida no algoritmo antes que este fosse compartilhado nesse ambiente supervisionado.

Essa implementação é adequada para diversos cenários e aplicações, sendo tratada primordialmente no artigo de Chow et al. (2003). Além dessa abordagem, existe também outro cenário, que utiliza uma chave dinâmica. No entanto, esse segundo cenário não será tratado por este trabalho.

Assim como demonstrado no capítulo anterior, em que foram apresentadas as tabelas de pesquisa, a abordagem adotada para esta dissertação parte do princípio que a chave criptográfica já foi inserida no algoritmo, sendo necessário desfazer algumas funções da implementação para conseguir resgatá-la. Essa abordagem segue o funcionamento da chave fixa, no entanto, esta também apresenta algumas debilidades.

Segundo Chow et al. (2003), um dos potenciais problemas do uso da chave fixa em uma determinada implementação, é que esta implementação pode ser extraída de seu contexto e utilizada para cifrar e decifrar criptogramas em um outro contexto, que utilize essa mesma chave fixa.

A fim de evitar que isto ocorresse, Chow et al. (2003) utilizaram recursos que alterassem o funcionamento padrão na entrada e saída do algoritmo de criptografia. Essa alteração consistia em substituir o algoritmo convencional, por um que tivesse entrada e saída codificadas. Sendo E_K a função encriptadora, esta seria substituída por $E'_K = G \circ E_K \circ F^{-1}$. Nesta nova função, F e G são codificações de entrada e saída, respectivamente.

Essas codificações são aleatórias e geradas de forma independente da chave K; maiores detalhes podem ser obtidos em (CHOW et al., 2003) . O uso dessas codificações visa dificultar a extração tanto da chave, quanto do próprio algoritmo de criptografia, para que este não seja utilizado integralmente em um outro contexto.

3.2.1 INTRODUZINDO AS CODIFICAÇÕES ENTRE AS TABELAS DE PESQUISA

Chow et al. (2003) verificaram que em cada etapa do processo de criptografia, haveria a possibilidade de que as informações manipuladas pudessem "vazar"; por vazar, entende-se

que os dados que fossem persistidos em cada etapa do processo de encriptação, mesmo que essa etapa fosse composta por tabelas de pesquisa aglutinadas, ainda ofereceriam informações suficientes para permitir que um determinado atacante pudesse inferir a chave criptográfica sem maiores dificuldades.

Visando atenuar este problema, Chow et al. (2003) resolveram adicionar essas codificações tanto nas entradas quanto nas saídas das tabela de pesquisa. Como cada uma das tabelas representavam etapas de armazenamento de informação na memória, elas poderiam persistir mais informações, além das alterações executadas pelas próprias funções do algoritmo. Dessa forma, cada tabela de pesquisa passou a apresentar o mesmo funcionamento descrito anteriormente para E'_K, que representava a função E_K entre as codificações F e G.

Estas codificações definidas para as bordas das **tabelas de pesquisa**, foram nomeadas como **codificações internas**, enquanto aquelas presentes na entrada e saída de **toda a implementação**, foram nomeadas como **codificações externas**.

3.2.2 CODIFICAÇÕES INTERNAS

Utilizando-se como modelo a implementação *WBAES*, fornecida por Chow et al. (2003) e tomando como exemplo a sua tabela de pesquisa que possui o maior número de aglutinações até o momento, $TBoxesTy_i Tables$, pode-se exemplificar o porquê apenas as funções do algoritmo não são suficientes para se proteger a chave criptográfica em um contexto *white-box*.

Supondo que o funcionamento dessa tabela, assim como sua execução para o *round* 1, sejam conhecidos por um determinado atacante. Este poderia considerar α como o *byte* da chave deste *round*, que está sendo inserido na $TBox$ que compõe a $TBoxTy_i Table$, T_0^1, e restringir as possibilidades de construção em função deste único *byte*. Conforme constatou Muir (2013), a partir de um único *byte*, existem 256 construções diferentes para essa $Ty_0 \circ T_0^1$,

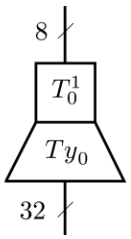

FIG. 3.1: $TBoxTy_i Table$, que mapeia 8 *bits* para 32 *bits*. Ilustração extraída de Muir (2013)

tendo em vista que para esses 8 *bits* de entrada, 2^8, correspondem 256 possibilidades. Pode-se facilmente enumerar as possíveis correspondências em uma lista, ou deduzir o resultado diretamente através do método apresentado por Muir (2013). Em que:

$$\alpha = S^{-1} \circ Ty_0^{-1} \circ (Ty_0 \circ T_0^1).$$

Neste caso, S^{-1} corresponde a inversa da operação *SubBytes*.

Logo, para proteger o conteúdo dessas tabelas de pesquisa, Chow et al. (2003) propuseram a utilização das codificações.

Conforme demonstrado para E_K, que ao adotar codificações em sua entrada e saída, tornou-se $E_K^{'} = G \circ E_K \circ F^{-1}$. Tem-se o mesmo comportamento para a tabela T nas codificações internas, que ao adotar as codificações g e f^{-1}(Sendo f a codificação de entrada e g a de saída) se tornará:

$$T^{'} = g \circ T \circ f^{-1}$$

Essa nova tabela, além de executar as operações de T e mapear entradas codificadas para saídas codificadas, deve conseguir também se conectar com outras tabelas, sem que nenhuma informação manipulada por T seja comprometida.

Para isso, ao realizar uma junção com outra tabela, esta nova estrutura deve preservar apenas as codificações mais externas, que estão na borda de cada uma dessas tabelas. Considere no exemplo abaixo, que a tabela T é sucedida por uma nova tabela R, que também possui codificações de entrada/saída.

$$R^{'} = h \circ T \circ g^{-1} \quad \text{e} \quad T^{'} = g \circ T \circ f^{-1}$$

A representação da tabela $R^{'} \circ T^{'}$ deve ser:

$$R^{'} \circ T^{'} = (h \circ R \circ g^{-1}) \circ (g \circ T \circ f^{-1}) = h \circ (R \circ T) \circ f^{-1}$$

Através dessas codificações, que devem ser canceláveis entre cada tabela de pesquisa, será possível preservar as conexões destas sem correr o risco de que reste alguma codificação pendente que corrompa o criptograma e impeça a sua reversão.

Devido a estrutura das tabelas XOR, projetadas por Chow et al. (2003), que são alimentadas por 8 *bits* em sua entrada, sendo 4 *bits* provenientes de uma tabela de pesquisa e os outros quatro de outra, Muir (2013) observou que quase todas as codificações utilizadas deveriam ser concatenadas e, a sua representação para adequar a quantidade de *bits* necessários para cada tabela, deveria iniciar primordialmente a partir uma codificação de

4 *bits*. Quando fosse necessário representar valores maiores do que esse, a solução seria a adoção de concatenações.

Por exemplo, para construir uma codificação *f*, de 8 *bits*, tem-se a seguinte representação a partir de f_0 e f_1, que possuem 4 *bits* cada:

$$f(x_0 \,||\, x_1) = f_0(x_0) \,||\, f_1(x_1)$$

O símbolo || representa a operação de concatenação. Esse símbolo será adotado quantas vezes se fizer necessário, para representar a quantidade de *bits* requisitada. Seguindo os exemplos de Muir (2013), para representar uma codificação de 32 *bits*, tem-se que:

$$f(x_0 \,||\, x_1 \,||\, ... \,||\, x_7) = f_0(x_0) \,||\, f_1(x_1) \,||\, ... \,||\, f_7(x_7)$$

3.2.2.1 SEGURANÇA LOCAL

Quando analisadas individualmente, as tabelas de pesquisa, que agora possuem codificação, apresentavam um comportamento que as tornavam teoricamente seguras, pelo menos do ponto de vista computacional. Estas não forneciam qualquer informação sobre a chave criptográfica a partir de sua saída e portanto, cumpriam o proposto pela confusão.

Para alcançar esse resultado, as codificações foram aplicadas a cada *byte* da chave, que antes apresentavam 256 saídas possíveis, em função de seus 8 *bits* de entrada, para cada *TBox*. Contudo, através da codificação, um **único** *byte* de entrada, 8 *bits*, dependendo de sua codificação, pode corresponder a 256 saídas diferentes, enquanto 256 diferentes *bytes* de entrada, podem corresponder a uma única saída.

Utilizando uma notação em forma de conjuntos, em que as possíveis entradas de uma determinada função são representadas pelo conjunto **A**, enquanto as possíveis saídas dessa função são representadas pelo conjunto **B**; pode-se considerar o seguinte:

Dados 8 *bits* de entrada dessa função, **F5**, por exemplo; tem-se que, de acordo com a codificação que modifica **F5**, sendo este um elemento do conjunto **A**, pode-se obter uma correspondência distinta dentre as 256 possíveis do conjunto **B**; conforme exemplificado na figura 3.2.

Da mesma forma, todos os 256 elementos do conjunto **A**, após terem sido modificados pela codificação, podem apresentar uma única correspondência no conjunto **B**; conforme a figura 3.3.

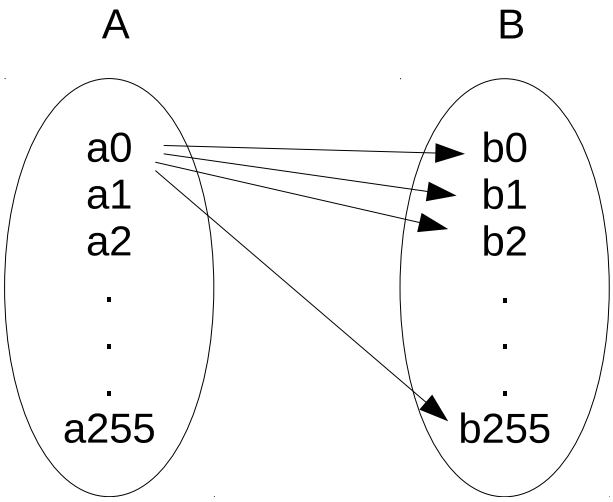

FIG. 3.2: Um elemento do conjunto A, que apresenta um correspondente diferente no conjunto B a cada nova codificação que o modifica.

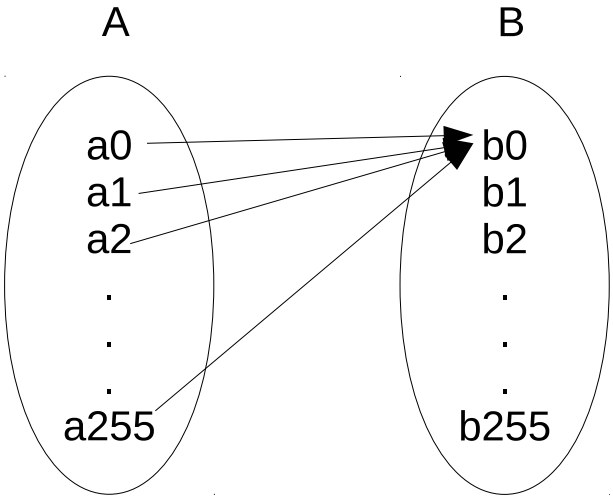

FIG. 3.3: Um elemento do conjunto B pode ser a correspondência de todos os elementos do conjunto A, que de acordo com suas respectivas codificações, podem apresentar o mesmo valor.

3.2.3 CODIFICAÇÕES EXTERNAS

No início desta seção foi apresentada a função criptográfica E_K e sua variação codificada, E'_K. As codificações externas têm esse nome por serem aplicadas de forma externa ao algoritmo de criptografia, sendo importante tanto para evitar que este seja extraído de seu ambiente de execução, quanto para que suas etapas não seja analisadas de forma isolada.

Como demonstrado por Beimers (2014), os tipos de codificações externas variam de acordo com o desenvolvedor que as implementam. Em seu artigo: "*White-box cryptography*" são demonstrados três modos de utilizar a codificação externa. O primeiro, sugerido por Chow et al. (2003), consiste na utilização de bijeções misturadoras de 128x128

bits, essas bijeções serão demonstradas no próximo capítulo, que trata sobre a difusão. O segunda, demonstrada por Muir (2013), é utilizar uma série de codificações concatenadas, como realizado nas codificações internas, até se alcançar os 128 *bits* desejados,

$$F = F_0 \,||\, F_1 \,||\, ... \,||\, F_{15} \text{ e } G = G_0 \,||\, G_1 \,||\, ... \,||\, G_{15}$$

e o terceiro modo, adotado na implementação do próprio Beimers (2014), consiste em realizar um XOR com um *array* de *bytes* de 128 *bits*, que corresponde a F e G.

3.3 UMA *SBOX* COM VALORES DEPENDENTES DA CHAVE

Segundo Shannon (1949), um algoritmo de criptografia simétrica deveria possuir critérios que tornassem ineficientes uma análise estatística realizada sobre esse determinado algoritmo. Desses critérios, pode-se citar a confusão e a difusão, sendo aquele introduzido principalmente por funções não-lineares, como funções de substituição, a exemplo do *SubBytes*, e este, introduzido por funções lineares, como *ShiftRows* e *MixColumns*.

A operação *SubBytes* executa a substituição dos *bytes* em função da matriz *Sbox*, que contém os valores de referência para substituição dos *bytes*. No entanto, esses valores são estáticos e públicos, ou seja, após uma substituição de *bytes* pontual, realizada sobre uma matriz do *state*, texto claro, ou k, chave criptográfica, não há necessidade de se conhecer a chave para resgatar os *bytes* que antecedem a substituição.

Uma das principais criptoanálises realizadas sobre a implementação *white-box* de Chow et al. (2003) e, que consegue com sucesso extrair cada chave da rodada em 2^{24} passos, é o *BGE-Attack*. Esta foi introduzida por Billet et al. (2005) e explora diversas debilidades do AES quando esse é adaptado para o ambiente *white-box*.

Uma dessas debilidades é a utilização de valores estáticos durante as transformações realizadas sobre o *state*. Esses valores são utilizados tanto pela função *SubBytes*, quanto pela função *MixColumns*. No entanto, nesta seção será tratada apenas a função *SubBytes*, tendo em vista que esta é a responsável por introduzir confusão ao algoritmo de criptografia.

As soluções propostas para aprimorar a implementação *white-box*, de forma que esta resistisse ao *BGE-Attack*, foram apresentadas primeiramente por Klinec et al. (2013) e posteriormente reforçadas e implementadas por Bačinská (2015).

Uma das soluções propostas por Klinec et al. (2013) e implementada por Bačinská (2015), é a adoção de elementos do *Twofish* em um contexto *white-box*.

O *Twofish* é um algoritmo de criptografia simétrica, projetado por Schneier et al. (1998), e foi um dos 5 finalistas do concurso do AES. Este capítulo tratará especificamente das características da *SBox* do *Twofish*, que possui valores de substituição definidos dinamicamente em função da chave, ao contrário do algoritmo vencedor do concurso AES, *Rijndael*, que tem os valores de sua *Sbox* estáticos e públicos.

3.3.1 A UTILIZAÇÃO DE ELEMENTOS DO TWOFISH EM *WHITE-BOX*

Desde a concepção da implementação *white-box*, o algoritmo que vem apresentando um maior volume de publicações que relacionam a sua arquitetura com o contexto *white-box*, é o AES.

Tendo sido introduzido a esse contexto por Chow et al. (2003), sua implementação para esse ambiente foi nomeada como *WBAES*. Essa nova implementação sofreu algumas melhorias em sua estrutura, no entanto, mesmo após essas alterações, esse algoritmo ainda se mostrou vulnerável para o contexto *white-box*, apresentando diversas debilidades de segurança.

Essas debilidades foram especialmente exploradas pelo *BGE-Attack*, como dito anteriormente.

Para realizar este ataque, a criptoanálise explorava pontos fracos que ainda não haviam sido tratados por Chow et al. (2003). Um deles, era que algumas operações do AES utilizavam valores constantes e publicamente conhecidos em seu processo. Dentre essas operações, estava o *SubBytes*, que é justamente o responsável por realizar a substituição de *bytes* de acordo com a matriz *Sbox*.

Essa debilidade, foi uma das identificadas por Klinec et al. (2013) e Bačinská (2015). Este último alterou a implementação do *WBAES* apresentada por Chow et al. (2003), visando combater esta e outras fraquezas remanescentes dessa implementação. Assim, foi gerado um novo algoritmo, nomeado por Bačinská (2015) como *WBAES+*.

Para aprimorar a confusão desse algoritmo, a implementação *white-box* de Bačinská (2015) fez uso das *Sboxes* do *Twofish*, que possuíam valores dinâmicos em função da chave. O funcionamento dessas *Sboxes* é apresentado abaixo:

$$
\begin{aligned}
S_0(\mathrm{x}) &= q_1[q_0[q_0[\mathrm{x}] \oplus s_{0,0}] \oplus s_{1,0}] \\
S_1(\mathrm{x}) &= q_0[q_0[q_1[\mathrm{x}] \oplus s_{0,1}] \oplus s_{1,1}] \\
S_2(\mathrm{x}) &= q_1[q_1[q_0[\mathrm{x}] \oplus s_{0,2}] \oplus s_{1,2}] \\
S_3(\mathrm{x}) &= q_0[q_1[q_1[\mathrm{x}] \oplus s_{0,3}] \oplus s_{1,3}]
\end{aligned}
$$

O *Twofish* utiliza quatro *S-boxes* 8x8, representadas por S_0, S_1, S_2 e S_3. Além disto, para tornar cada *Sbox* dinâmica são utilizadas duas subchaves de 32 *bits*. Os *bytes* dessas subchaves são representados por $s_{i,0}$, $s_{i,1}$, $s_{i,2}$ e $s_{i,3}$, onde i assume os valores 0 e 1.

Esses *bytes*, oriundos das subchaves mencionadas acima, são gerados a partir de um processo de expansão de chave, que será discutido no capítulo 4. Já os *bytes* q_0 e q_1, correspondem aos *bytes* do *state*, após estes sofrerem algumas transformações. Essas transformações podem ser observadas no artigo de Schneier et al. (1998): "*Twofish: A 128-Bit Block Cipher*", na seção 4.3.5, *The Permutations qo and q1.*

Na implementação original do *Twofish*, os *bytes* que saem da *Sbox* são interpretados como uma matriz 4x1 *bytes*, ou vetor coluna e em seguida multiplicados por uma matriz denominada *MDS*, *Maximum Distance Separable*, que por ser responsável por introduzir difusão ao algoritmo, será tratada no próximo capítulo.

4 DIFUSÃO

A difusão consiste basicamente em "espalhar" os padrões existentes no texto claro, M, para que esses padrões não sejam reproduzidos no criptograma. Embora Shannon (1949) acredite que não há como impedir que o atacante estabeleça uma análise estatística, a menos que se faça uso da compressão, este considera que a difusão faz com que a redundância seja dissipada através da estrutura que leva M até E, que é o criptograma; tornando mais complexa a relação existente entre M e E, fazendo com que a alteração de cada *byte* do texto claro impacte em diversos *bytes* do criptograma e vice-versa.

4.1 FUNÇÕES RESPONSÁVEIS POR INTRODUZIR DIFUSÃO

Na implementação do AES, que é o algoritmo que possui mais trabalhos relacionados ao contexto *white-box*, tem-se o *MixColumns* em conjunto com *ShiftRows* como as operações responsáveis por introduzirem difusão ao algoritmo. No entanto, como em um ambiente *white-box* o acesso a memória é efetuado após a realização de um único *round*, a baixa taxa de difusão introduzida pelo *MixColumns* + *ShiftRows* não é suficiente para estabelecer uma relação complexa entre os *bytes* de entrada e saída do *round*.

Considerando uma transformação individual realizada pelo *MixColumns*, conforme apresenta a figura 4.1, pode-se atestar que cada *byte* de saída é dependente de apenas 4 *bytes* de entrada e da mesma forma, cada *byte* de entrada afeta apenas 4 *bytes* na saída do *round*; não se alcançando o desejado pelo efeito avalanche[5] para um único *round*.

Além da baixa difusão, quando analisada individualmente para um determinado round, a função *MixColumns* também faz uso de uma matriz com valores estáticos e publicamente conhecidos. Essas debilidades foram algumas das exploradas pelo *BGE-Attack* para extrair a chave da implementação do WBAES.

Devido a série de mudanças que o WBAES deveria sofrer para que fosse possível resistir ao *BGE-Attack* ou a outras criptoanálises que explorassem as mesmas debilidades, Bačinská (2015) resolveu romper a compatibilidade existente com o AES(WBAES) e criar uma nova cifra que, mesmo fazendo uso de diversas funções do AES, assim como de funções e conceitos introduzidos por Chow et al. (2003) para o WBAES, não se preocupava em

[5]O efeito avalanche, introduzido por Feistel et al. (1975), estabelece como uma propriedade desejável para algoritmos criptográficos que, uma pequena mudança nos *bits* de entrada deve afetar todos ou muitos *bits* de saída

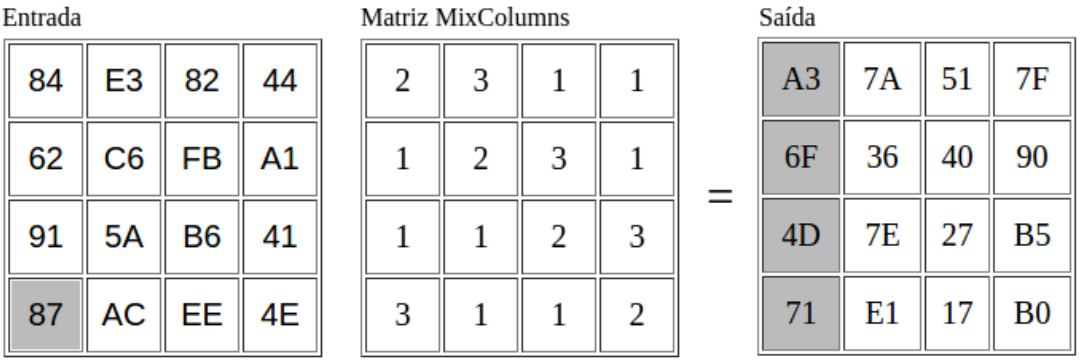

FIG. 4.1: A alteração de um *byte* na entrada, afeta um única coluna na saída, devido ao funcionamento do *MixColumns*.

gerar o mesmo criptograma e adotava alguns métodos distintos dos utilizados por esses algoritmos.

Assim, foi gerado o WBAES+, que visando aprimorar a difusão, propunha a substituição da matriz *MixColumns* e *ShiftRows* por um outro tipo de matriz MDS(Maximum Distance Separable), que atuasse sobre todo o *state*, fazendo com que qualquer *byte* que fosse alterado na entrada da matriz, impactasse todos os *bytes* de saída após um único *round*, oferecendo dessa forma, maiores encargos a análise de um possível atacante.

4.2 A MATRIZ MDS

Um código linear[n, k, d] sobre GF(2^p) é um subespaço com dimensão k do espaço vetorial $(\mathrm{GF}(2^p))^n$, onde a distância de Hamming, número de posições em que dois vetores diferem, entre dois n-elementos distintos é pelo menos e no máximo igual a d.

Esses códigos satisfazem o limite de Singleton, em que $d \leq n - k + 1$. Uma codificação linear que atinge a igualdade neste limite é chamada de codificação MDS, conforme demonstra Singleton (1964).

Segundo Abrahao (2007), as matrizes MDS têm tido importante papel no projeto de diversas cifras de bloco como as SHARK(Rijmen et al. (1996)), Square(Daemen et al. (1997)) e Rijndael(Daemen e Rijmen (2000)), que é o AES; introduzindo difusão ao algoritmo, sem a necessidade de se realizar grandes rodadas neste.

Uma das maneiras de se obter matrizes MDS é através da utilização de matrizes circulares, este é o caso do AES, como será demonstrado abaixo; em que cada linha é uma instância rotacionada da primeira. Na operação *MixColumns*, por exemplo, tem-se a utilização de uma matriz que, apesar de MDS, não é involutória; pois, considerando-se A como a matriz do AES, um produto desta por ela mesma, deveria resultar em I_{4x4}, que

corresponde a matriz identidade: $A \cdot A = I_{4x4}$. Isto é exemplificado pela figura 4.2.

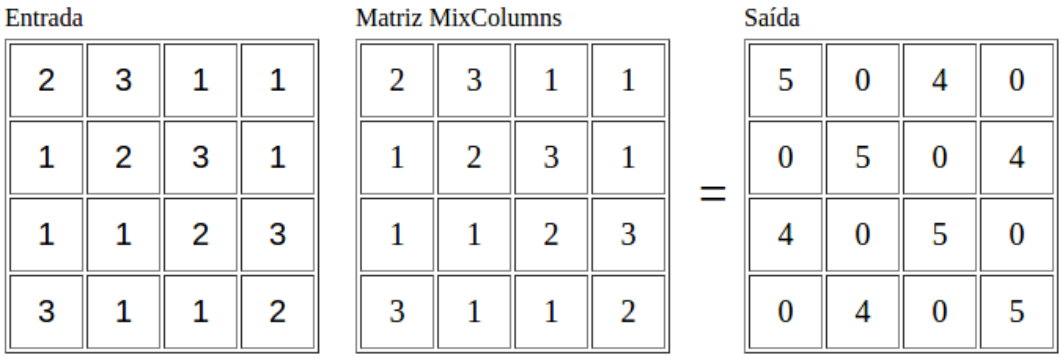

FIG. 4.2: Matriz MixColumns que, apesar de MDS, não é involutória, pois não retorna a identidade quando multiplicada por si.

A fim de utilizar matrizes que introduzissem uma completa difusão em único round e ainda fossem involutórias, Bačinská (2015) resolveu utilizar a aplicação das matrizes MDS ao AES, que havia sido proposta por Abrahao (2007) e nomeadas por este como MDS-AES.

O MDS-AES sugere a substituição das operações MixColumns e ShiftRows, que são as responsáveis por adicionar difusão ao algoritmo AES, por matrizes MDS de dimensão 16x16, como a demonstrada na figura 4.3. Apesar das funções MixColumns e Shiftrows introduzirem difusão ao algoritmo, como foi abordado anteriormente, estas não o realizam de forma completa em um único round; fazendo com que nem todos os *bytes* de saída tenham dependência de todos os *bytes* de entrada.

A técnica utilizada por Abrahao (2007) para a construção de matrizes involutórias envolve as denominadas matrizes Cauchy, que segundo ele, é também utilizadas em algumas cifras de bloco como Khazad(Barreto e Rijmen (2000)) e Anubis(Rijmen e Barreto (2000)).

4.2.1 MATRIZES DE CAUCHY

Pelo fato das matrizes de Cauchy, além de involutórias, em que a matriz é sua própria inversa, ainda apresentarem outras características desejáveis para o MDS-AES de Abrahao (2007), como por exemplo: Serem MDS, terem a capacidade de produzir matrizes com dimensão 16x16, apresentarem em cada um dos elementos de sua matriz, baixo peso de Hamming e etc. Este considerou determinante a sua adoção.

Em função disto, Abrahao (2007) implementou um gerador desse tipo de matrizes, utilizando a linguagem Java. Esse gerador, denominado por ele como GMC(Gerador

de Matrizes de Cauchy), segue uma série de restrições; demonstradas no capítulo 5 dessa dissertação, em que são abordadas as funções dependentes de chave no algoritmo. Segundo Abrahao (2007), a melhor matriz seguindo essas restrições, é a apresentada na figura 4.3. Essa matriz é utilizada como uma alternativa, caso não seja possível gerar uma matriz dependente de chave; este processo também é exemplificado no capítulo 5.

$$M_{16\times16} = \begin{bmatrix}
01_x & 03_x & 04_x & 05_x & 06_x & 07_x & 08_x & 09_x & 0a_x & 0b_x & 0c_x & 0d_x & 0e_x & 10_x & 02_x & 1e_x \\
03_x & 01_x & 05_x & 04_x & 07_x & 06_x & 09_x & 08_x & 0b_x & 0a_x & 0d_x & 0c_x & 10_x & 0e_x & 1e_x & 02_x \\
04_x & 05_x & 01_x & 03_x & 08_x & 09_x & 06_x & 07_x & 0c_x & 0d_x & 0a_x & 0b_x & 02_x & 1e_x & 0e_x & 10_x \\
05_x & 04_x & 03_x & 01_x & 09_x & 08_x & 07_x & 06_x & 0d_x & 0c_x & 0b_x & 0a_x & 1e_x & 02_x & 10_x & 0e_x \\
06_x & 07_x & 08_x & 09_x & 01_x & 03_x & 04_x & 05_x & 0e_x & 10_x & 02_x & 1e_x & 0a_x & 0b_x & 0c_x & 0d_x \\
07_x & 06_x & 09_x & 08_x & 03_x & 01_x & 05_x & 04_x & 10_x & 0e_x & 1e_x & 02_x & 0b_x & 0a_x & 0d_x & 0c_x \\
08_x & 09_x & 06_x & 07_x & 04_x & 05_x & 01_x & 03_x & 02_x & 1e_x & 0e_x & 10_x & 0c_x & 0d_x & 0a_x & 0b_x \\
09_x & 08_x & 07_x & 06_x & 05_x & 04_x & 03_x & 01_x & 1e_x & 02_x & 10_x & 0e_x & 0d_x & 0c_x & 0b_x & 0a_x \\
0a_x & 0b_x & 0c_x & 0d_x & 0e_x & 10_x & 02_x & 1e_x & 01_x & 03_x & 04_x & 05_x & 06_x & 07_x & 08_x & 09_x \\
0b_x & 0a_x & 0d_x & 0c_x & 10_x & 0e_x & 1e_x & 02_x & 03_x & 01_x & 05_x & 04_x & 07_x & 06_x & 09_x & 08_x \\
0c_x & 0d_x & 0a_x & 0b_x & 02_x & 1e_x & 0e_x & 10_x & 04_x & 05_x & 01_x & 03_x & 08_x & 09_x & 06_x & 07_x \\
0d_x & 0c_x & 0b_x & 0a_x & 1e_x & 02_x & 10_x & 0e_x & 05_x & 04_x & 03_x & 01_x & 09_x & 08_x & 07_x & 06_x \\
0e_x & 10_x & 02_x & 1e_x & 0a_x & 0b_x & 0c_x & 0d_x & 06_x & 07_x & 08_x & 09_x & 01_x & 03_x & 04_x & 05_x \\
10_x & 0e_x & 1e_x & 02_x & 0b_x & 0a_x & 0d_x & 0c_x & 07_x & 06_x & 09_x & 08_x & 03_x & 01_x & 05_x & 04_x \\
02_x & 1e_x & 0e_x & 10_x & 0c_x & 0d_x & 0a_x & 0b_x & 08_x & 09_x & 06_x & 07_x & 04_x & 05_x & 01_x & 03_x \\
1e_x & 02_x & 10_x & 0e_x & 0d_x & 0c_x & 0b_x & 0a_x & 09_x & 08_x & 07_x & 06_x & 05_x & 04_x & 03_x & 01_x
\end{bmatrix}$$

FIG. 4.3: Matriz MDS 16x16, apresentada por Abrahao (2007)

4.3 BIJEÇÕES MISTURADORAS

Apesar da seção anterior apresentar um recurso importante para aumentar a difusão do algoritmo, que trata principalmente da expansão da matriz MDS de 4x4 *bytes* para 16x16 *bytes*, uma outra funcionalidade para aprimorar a difusão já havia sido introduzida anteriormente ao proposto por Klinec et al. (2013) e, posteriormente, por Bačinská (2015). Essa proposta anterior consistia nas bijeções misturadoras e foram adotadas por Chow

et al. (2003) em seu artigo pioneiro da implementação *white-box*.

Quando Chow et al. (2003) realizaram seu conjunto de propostas visando tornar o AES resistente as debilidades do contexto *white-box*, eles inicialmente apresentaram a restruturação das funções originais do algoritmo em tabelas de pesquisa, para que dessa forma estas pudessem minimizar o armazenamento de informações que levassem um atacante diretamente até a chave, a partir da memória; contudo, o objetivo principal foi a utilização tantos das codificações quanto das bijeções misturadoras.

As codificações foram demonstradas no capítulo que tratou sobre confusão, sendo abordadas as codificações internas e as codificações externas, ambas não lineares. Já esta seção trata das transformações lineares, em específico, as bijeções misturadoras que, foram a primeira tentativa de adquirir um aprimoramento na difusão do algoritmo em um contexto *white-box*.

4.3.1 DEFININDO AS BIJEÇÕES MISTURADORAS

As bijeções misturadoras são matrizes invertíveis sobre GF(2). O processo de criação dessas matrizes pode ser dado através da geração de uma matriz aleatória, sendo em seguida verificado se essa matriz é invertível e, caso esta não seja, deve-se repetir o processo, até que se obtenha o resultado esperado. Segundo os cálculos de Muir (2013), o número médio de tentativas para se obter uma matriz 32x32 *bits* que atendesse a esse requisito, era de 3.47 iterações; Chow et al. (2003), em sua seção sobre as bijeções misturadoras, citam a proposta de Xiao e Zhou (2002) como uma outra forma de gerar essas matrizes.

As bijeções misturadoras são inseridas imediatamente nas extremidades das tabelas de pesquisa em que há dependência de chave, estando presente entre as tabelas de pesquisa e as codificações, que foram citadas no capítulo sobre confusão.

No caso da implementação *white-box*, apresentada por Chow et al. (2003), a dependência de chave ocorre nas tabelas *Tboxes* e nas *TBoxesTyiTables* ou tabelas compostas, que continham a função *AddRoundKey*, como foi demonstrado no capítulo 2.

A fim de atender as tabelas de pesquisa que projetou, Chow et al. (2003) verificaram que seria necessário criar dois tipos tipos de bijeção: Uma de 8x8 *bits*, que estaria na entrada das tabelas de pesquisa, denominada L; e outra de 32x32 *bits*, denominada MB.

A utilização dessas bijeções nos *rounds* internos, que compreendiam do *round* 2 até o 9, seria a mesma; no entanto, para os *rounds* 1 e 10, não haveria bijeção de entrada, e nem bijeção de saída, respectivamente; isto é devido às codificações externas que foram apresentadas no capítulo 3.

Sendo necessário um total de: 144 bijeções L, 16 bijeções para cada um dos 9 *rounds*, tendo em vista que o primeiro round não apresenta essas bijeções; 36 bijeções MB, 4 bijeções para cada um dos 9 *rounds*, já que estas não são aplicadas ao último round.

4.3.2 O FUNCIONAMENTO DAS BIJEÇÕES

Para melhor entender o funcionamento das bijeções, é demonstrado a seguir o comportamento dessas transformações, de forma similar a abordada por Muir (2013) em seu artigo; que inicia apresentando as tabelas de pesquisa com dependência de chave e, posteriormente, inclui as bijeções de entrada e saída em cada uma dessas tabelas, até que se atinja o projeto final do round. A seguir são apresentadas as 4 primeiras tabelas de pesquisa, com dependência de chave, para o round 2.

$$Ty_0 \circ T_0^2,$$
$$Ty_1 \circ T_1^2,$$
$$Ty_2 \circ T_2^2,$$
$$Ty_3 \circ T_3^2.$$

Sendo as tabelas acima equivalentes às quatro primeiras *TBoxesTyiTables* ou tabelas compostas, do round 2 e, tendo em vista que essas consistem em tabelas com entrada de 8 *bits* e saída de 32 *bits*; tem-se que, para essas quatro primeiras tabelas, são necessárias 4 bijeções de entrada de 8x8 *bits*: L_0^2, L_1^2, L_2^2 e L_3^2; além de MB, que é a bijeção de saída de 32x32 *bits*, escolhida para essas quatro tabelas. Produzindo as seguintes tabelas:

$$MB \circ Ty_0 \circ T_0^2 \circ L_0^{2^{-1}}$$
$$MB \circ Ty_1 \circ T_1^2 \circ L_1^{2^{-1}}$$
$$MB \circ Ty_2 \circ T_2^2 \circ L_2^{2^{-1}}$$
$$MB \circ Ty_3 \circ T_3^2 \circ L_3^{2^{-1}}$$

Verifica-se que, as bijeções de entrada dessas tabelas do round 2, são inversas das bijeções L_0^2, L_1^2, L_2^2 e L_3^2, isto porquê, essas foram as bijeções utilizadas no round 1 para computar os *bytes* de entrada dessas tabelas de pesquisa no round 2; fazendo com que as bijeções se anulem.

Após este processo, as saídas dessas tabelas serão conectadas em três níveis de tabelas XOR, que ao fim do terceiro nível, retornará 32 *bits*, conforme mostra a figura 4.4.

O resultado desse processo implica no seguinte: $MB \circ MC\ [z_0 z_1 z_2 z_3]^T$.

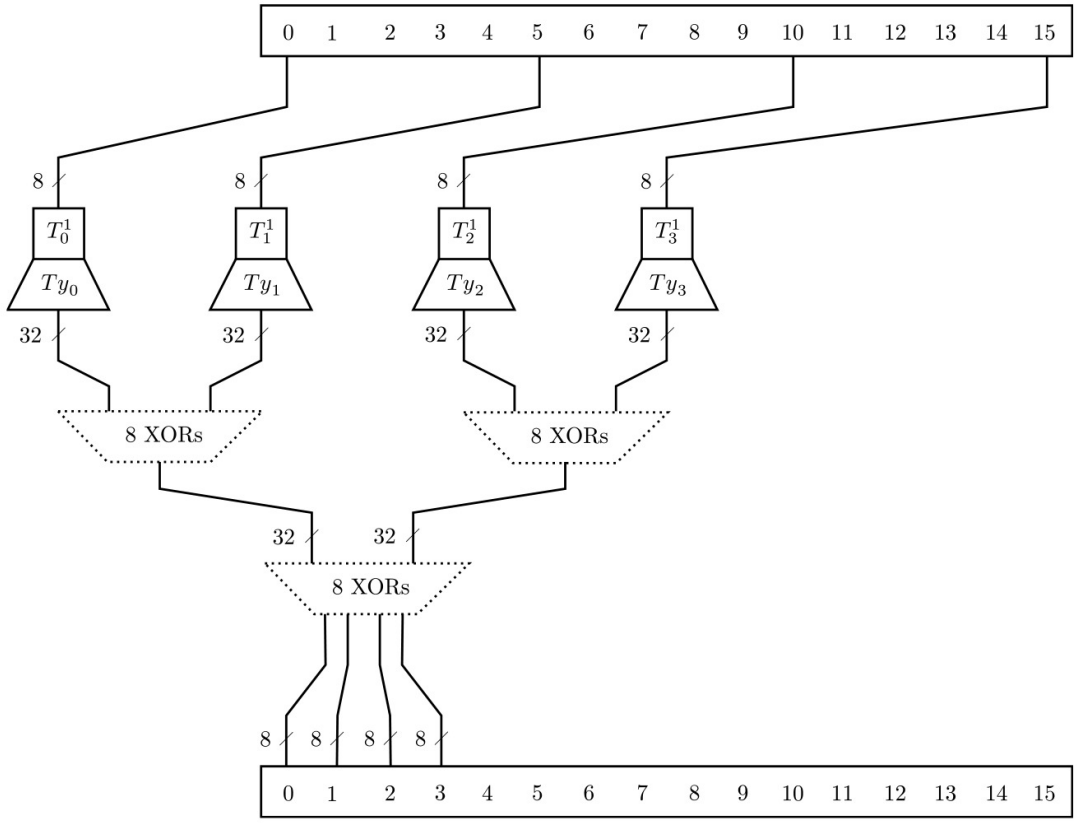

FIG. 4.4: Fluxo de dados para o round 1 do AES, com o *state* de entrada ao topo e o de saída na parte inferior. Ilustrado por Muir (2013)

Sendo MB a bijeção remanescente, o objetivo agora é removê-la. Essa bijeção é do tipo 32x32 *bits*; após a remoção desta, será preciso aplicar as bijeções de entrada 8x8 *bits*, previstas para o próximo *round*.

Para remover a bijeção 32x32 *bits*, Muir (2013) demonstrou a necessidade primeira de se aplicar uma técnica de decomposição de matrizes sobre MB^{-1}, que é a inversa de MB. O resultado foram quatro novas matrizes de 8x32 *bits*, conforme a figura 4.5.

$$MB^{-1}\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = MB^{-1}\begin{bmatrix} z_0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ z_1 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ z_2 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ 0 \\ z_3 \end{bmatrix}.$$

FIG. 4.5: Decomposição da bijeção misturadora em quatro tabelas de 8x32 *bits*. Ilustrado por Muir (2013)

Sendo MB_0^{-1}, MB_1^{-1}, MB_2^{-1} e MB_3^{-1} correspondentes as quatro matrizes 8x32 *bits*, demonstrada na figura 4.5; na saída dessas matrizes, tem-se L^3, uma bijeção 32x32 *bits*,

construída a partir da concatenação das quatro bijeções de entrada de 8x8 *bits* do terceiro round. Conforme é demonstrado abaixo:

$$L^3 = L_0^3 \,||\, L_{13}^3 \,||\, L_{10}^3 \,||\, L_7^3$$

L^3 aplica as respectivas codificações aos *bytes* das posições 0, 1, 2 e 3, que estão sendo introduzidos no terceiro *round*; sendo a definição realizada acima para L^3, posterior a transformação *ShiftRows*, o que impactou o índice dos *bytes* entrantes. Conforme mencionado, a bijeção L^3 é colocada na saída de cada uma das bijeções MB_i^{-1}, resultando no seguinte:

$$L^3 \circ MB_0^{-1},$$
$$L^3 \circ MB_1^{-1},$$
$$L^3 \circ MB_2^{-1},$$
$$L^3 \circ MB_3^{-1}$$

As saídas dessas tabelas são combinadas através de diversas tabelas XOR. Esse processo é demonstrado na figura 4.6 e, se comparada à figura 4.4, percebe-se que as tabelas dobraram.

No início da seção que tratou sobre as bijeções, foi exemplificado tanto a quantidade de tabelas quanto que as bijeções não seriam completamente aplicadas nem ao primeiro e nem ao último round, não havendo bijeções de saída para o primeiro e nem bijeções de entrada para o último. Deste modo, pode-se considerar a aplicação demonstrada na figura 4.6 para todos os *rounds* do *WBAES* de Chow et al. (2003), sendo necessário apenas que se considere as exceções citadas, que tratam sobre as bijeções de entrada e saída nos *rounds* 1 e 10.

Uma outra coisa importante de se pontuar é que ao contrário das matrizes que foram tratadas no início desse capítulo, essas bijeções não geram impacto sobre o criptograma do algoritmo, fazendo com que este se mantenha o mesmo, já que essas se anulam em cada um dos rounds. Contudo, no projeto do *WBAES*, estas não foram suficientes para se alcançar o efeito avalanche em um único *round*.

Além disto, é necessário acrescentar que o *WBAES+*, desenvolvido por Bačinská (2015), expandiu as bijeções misturadoras, fazendo com que estas se tornassem mapeadores de 8x128 *bits*, ao invés de 8x32 *bits*; que eram utilizados anteriormente no *WBAES*, de modo que acompanhassem as matrizes MDS que foram ampliadas.

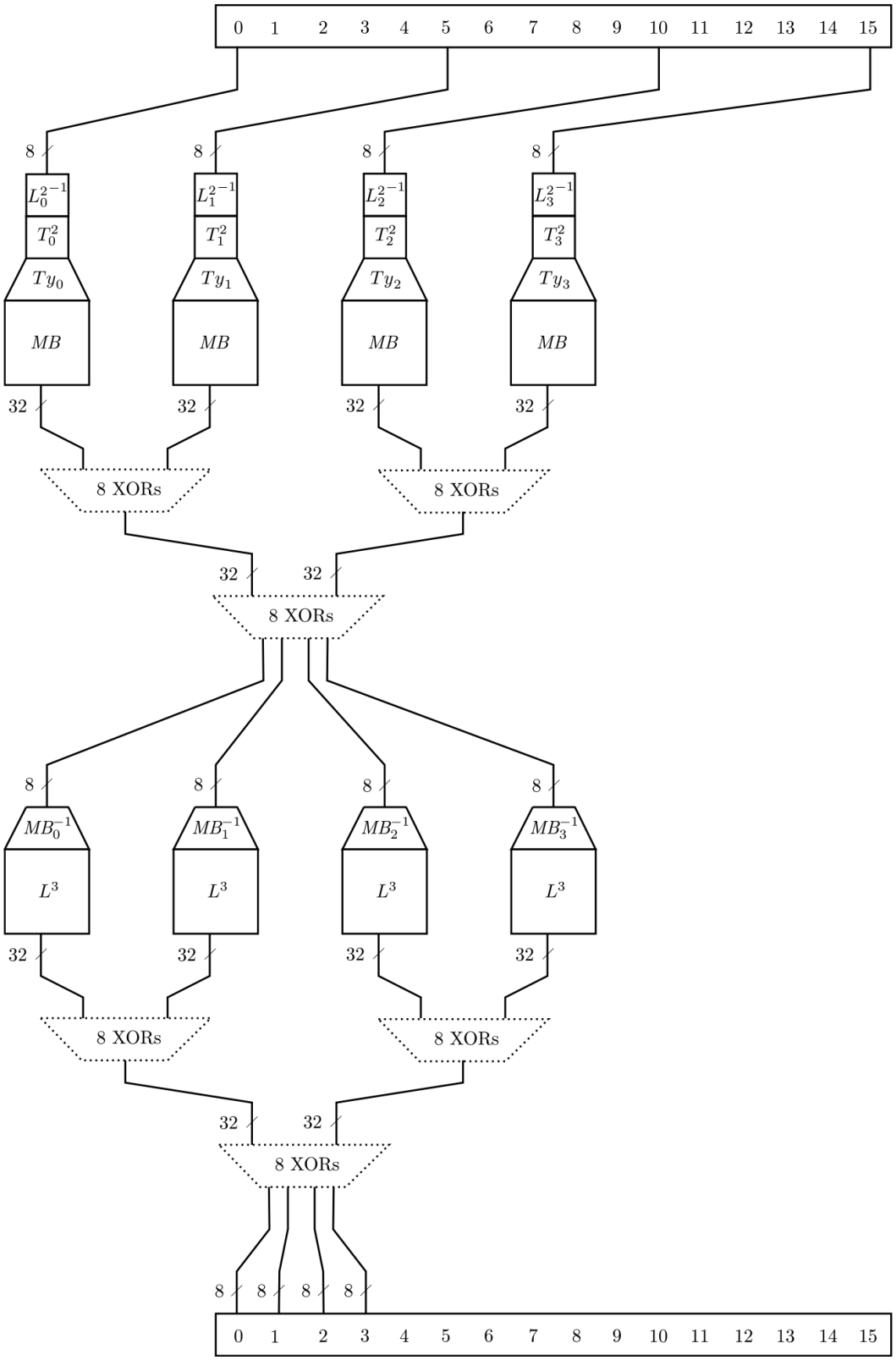

FIG. 4.6: Aplicação das bijeções misturadoras no *round* 2. A representação do *round* 3 ao 9 é a mesma. Para os rounds 1 e 10, a única diferença é a ausência de bijeções misturadoras na entrada das *Tboxes* para o *round* 1 e na saída das *Tboxes* para o *round* 10. Ilustrado por Muir (2013).

5 A CHAVE CRIPTOGRÁFICA

Nos capítulos anteriores foram demonstradas diversas técnicas para aprimorar a segurança do algoritmo, de modo que o vazamento de informações fosse minimizado, a fim de não permitir inferência da chave criptográfica. No entanto, ainda mais importante do que assegurar as funções do algoritmo, que manipulam a chave em conjunto com o *state* para produção do criptograma, é tornar os processos que manipulam essa chave diretamente, mais seguros.

Um dos procedimentos utilizados para permitir que instâncias diferentes de uma mesma chave possam ser utilizadas no processo de criptografia, é o uso da expansão. A expansão permite a partir de uma chave inicial, gerar tantas chaves quanto necessário. Normalmente, a quantidade de chaves a serem geradas são determinadas de acordo com o número de *rounds* que aquele algoritmo possui.

O WBAES utiliza um processo de expansão de chave que foi originalmente projetado para o algoritmo AES, em um contexto *black-box*. Contudo, este processo de expansão de chave se mostrou vulnerável quando implementado em um contexto *white-box*. Nesse contexto, o *BGE-Attack*, projetado por Billet et al. (2005), pôde explorar as vulnerabilidades existentes na expansão, para que a partir de duas chaves de *rounds* consecutivos que fossem inferidas, deduzir a chave original.

Já na implementação idealizada por Bačinská (2015), que propunha o WBAES+ para solucionar alguns dos problemas existentes na implementação *white-box* de Chow et al. (2003), foi utilizada uma expansão de chave em que as chaves dos demais *rounds* eram geradas a partir de uma função *hash* utilizando a chave original.

Tendo em vista que as funções *one-way hash* foram projetadas para não serem invertíveis, a utilização destas em um contexto *white-box*, permite inviabilizar uma das etapas propostas pelo BGE-Attack citadas anteriormente, em que era possível inferir a chave original a partir da captura de duas chaves de *rounds* que fossem consecutivos.

5.1 A UTILIZAÇÃO DE FUNÇÕES *ONE-WAY HASH* PARA EXPANSÃO DA CHAVE

Ao tratar sobre a expansão de chave, Bačinská (2015) inicia apresentando o conceito de função de derivação de chave ou KDF(Key derivation function). Através da introdução dessa técnica, Bačinská (2015) explica que esta permite a partir de uma chave principal,

gerar outras chaves que serão utilizadas pelo algoritmo de criptografia.

A expansão de chave convencional do AES é considerada também um tipo de KDF, entretanto, as características buscadas por funções deste tipo em um contexto *white-box*, não são apresentadas pela função utilizada pelo AES, como é o caso das funções *hash*, que são funções matematicamente irreversíveis.

Uma outra característica muito importante buscada por Bačinská (2015), dentre as funções para se expandir a chave criptográfica, está relacionada ao custo computacional exigido por elas durante o seu processo de execução, visando tornar ataques de força bruta ou por dicionário, inviáveis.

O tipo de KDF adotado por Bačinská (2015) foi o *Scrypt*. Criado por Percival (2009), este foi desenvolvido considerando ataques com *hardware* personalizado em mente, se baseando no conceito de ***sequential memory-hard functions***.

Esse conceito é definido por Bačinská (2015) como: "Um recurso que visa combater a efetividade de um ataque de força bruta, através do consumo de uma grande quantidade de memória durante a execução do ataque"; além disso, esse recurso minimiza também a efetividade de um ataque de força bruta, mesmo seja utilizada uma técnica de paralelismo durante a execução.

5.1.1 O SCRYPT

Conforme introduzido anteriormente, o Scrypt foi criado por Percival (2009) e adotado por Bačinská (2015), como KDF em um contexto *white-box*. Uma das características que fizeram com que esse algoritmo fosse questionado como uma boa alternativa para o contexto *white-box*, foi o fato deste incorporar o *Sequential memory-hard functions*. Embora as implicações desse conceito tenham sido brevemente abordadas, é necessário que este seja tratado de forma mais detalhada; o que é realizado a seguir.

Em seu artigo criador do Scrypt, Percival (2009) apresentou uma série de definições que variavam desde o algoritmo *memory-hard*, definição 1, até a demonstração da função Scrypt, definição 4. Contudo, Bačinská (2015) ao apresentar o Scrypt, inicia demonstrando imediatamente a definição 2, que corresponde especificamente a *sequential memory-hard function* e, em seguida, apresenta os algoritmos integrantes da função do Scrypt, que equivale a definição 4.

No entanto, aqui será apresentada também a definição 1, que se refere ao algoritmo *memory-hard*. Introduzir essa definição, será importante para permitir uma melhor concatenação das ideias que embasam a compreensão das *sequential memory-hard functions*

e, consequentemente, do Scrypt.

Definição 1: Um algoritmo *memory-hard* em uma Random Access Machine é um algoritmo que utiliza espaço S(n) e T(n) operações, onde S(n) $\in \Omega(\mathrm{T}(n)^{1-\epsilon}$.

Um algoritmo *memory-hard* é, portanto, um algoritmo que, assintoticamente, utiliza quase tantas alocações de memória quanto suas operações, fazendo uso de aproximadamente toda a memória disponível para um determinado número de operações a serem executadas.

A partir dessa primeira definição, que compreende o algoritmo *memory-hard*, pode-se introduzir a classe de funções que, além de manipular a memória para dificultar a execução de ataques, visam impedir também aqueles ataques que fazem uso de paralelismo. Essa classe compreende a segunda definição:

Definiçao 2: Uma *sequential memory-hard function* é uma função que

(a) Pode ser computada por um algoritmo *memory-hard* em uma Random Access machine em *T(n)* operações; e

(b) Não pode ser computada em uma Parallel Random Access Machine com *S*(n)* processadores e espaço *S*(n)*, no tempo esperado *T*(n)*, onde $S^*(n)T^*(n) = \mathrm{O}(T(n)^{2-x})$ para qualquer x > 0.

Pode-se constatar então que, uma *sequential memory-hard function* implementa o conceito do algoritmo *memory-hard*, conforme apresentado no primeiro item da definição 2. Além disso, esta estende a aplicação do conceito de *memory-hard*, inviabilizando ataques que utilizem computação paralela e tornando, segundo Percival (2009), impossível para que um algoritmo paralelo, alcance assintoticamente, um baixo custo computacional que seja significativo.

5.2 AS PROPOSTAS DE KLINEC X AS IMPLEMENTAÇÕES DE BAČINSKÁ - A CHAVE EXPANDIDA PARA AS DIFERENTES FUNÇÕES EM *WHITE-BOX*

Grande parte dos recursos implementados pelo algoritmo *WBAES+*, proposto por Bačinská (2015), foram uma aplicação de alguns conceitos propostos inicialmente por Klinec et al. (2013). Nesta seção, será possível identificar que algumas vezes as dissertações desses dois autores se misturam e as referências utilizadas por estes, intercedem de forma natural.

5.2.1 CHAVES DE *ROUND*

Anteriormente, foi demonstrado que a expansão de chave adotada pelo AES, quando submetida ao BGE-Attack, poderia ter suas chaves inferidas através da captura de duas chaves de rounds que fossem consecutivos, permitindo-se obter a chave principal do algoritmo. Visando resolver este problema, Klinec et al. (2013) propôs a utilização de uma função *hash* para gerar as chaves do *round*, já que por sua característica, esse tipo de função não apresentaria um caminho inverso existente entre as chaves de *round* e a principal.

Dentre as funções elencadas por Klinec et al. (2013) para realizar essa expansão, foram sugeridas dois tipos de KDF, o *bcrypt* e o *scrypt*, ambos com características similares; conforme o apresentado na seção anterior, verificou-se que o *scrypt* foi o KDF adotado por Bačinská (2015) em sua implementação.

Entretanto, além do KDF, Bačinská (2015) também seguiu as especificações de Klinec et al. (2013) para a geração das chaves que seriam utilizadas para encriptar os *rounds*, produzir as *sboxes* e as matrizes MDS.

Nessa subseção, verificar-se-á os procedimentos que implicaram na geração das chaves de cada *round* e, será comparado esse processo de geração com o utilizado pelas funções Sbox e MDS.

A função responsável por gerar as chaves para cada um dos *rounds*, é apresentada na figura 5.1. Esta foi extraída de Bačinská (2015) e foi originalmente apresentada por Klinec et al. (2013):

$$k^r = \begin{cases} hash_{sc\_N,sc\_r,sc\_p,n\_sha}(key, salt) & \text{if } r = 0 \\ hash_{sc\_N,sc\_r,sc\_p,n\_sha}(k^{r-1}||key, salt) & \text{otherwise} \end{cases}$$

FIG. 5.1: Expansão de chave para encriptar os rounds do algoritmo, proposta por Klinec et al. (2013).

Para essa função, os parâmetros passados são descritos a seguir:

- *key* - Chave criptográfica de 128 *bits*.
- k^r - Chave correspondente ao round r
- || - Símbolo de concatenação de valores
- *salt* - Esse *salt* corresponde a uma *string* de 128 *bits*, tendo sido inicialmente sugerida por Klinec et al. (2013) como um valor publicamente conhecido ou não, que poderia

ser fruto do texto claro ou gerado aleatoriamente durante o processo de encriptação. Já Bačinská (2015), nesse caso, optou pelo uso de uma *string* publicamente conhecida, que possuía o valor estático igual a "TheConstantString."

Tanto Klinec et al. (2013) quanto Bačinská (2015), a fim de assegurar a incapacidade do atacante, ao determinar a chave de um *round* a partir da derivação da chaves dos demais *rounds*, resolveram concatenar a chave criptográfica em conjunto com a chave do *round* anterior. Nesse caso, todos os *rounds*, exceto o primeiro, no processo de geração da chave do *round* corrente, recebem o resultado da função *scrypt* aplicada a uma concatenação entre a chave principal e a chave do round anterior.

5.2.2 UMA *SBOX* COM DEPENDÊNCIA DE CHAVE

Como foi observado no capítulo 2, que tratou especificamente sobre a confusão em um contexto *white-box*; as implicações de uma *sbox* estática pode fazer com que a implementação do algoritmo esteja mais suscetível a ataques de criptoanálise, como o BGE-Attack, por exemplo.

Por isto, nesse capítulo que tratou da confusão, foi apresentada a utilização de uma *sbox* dependente da chave criptográfica, essa *sbox* era a prevista para o algoritmo de criptografia Twofish. No entanto, apesar de dependente da chave criptográfica, Klinec et al. (2013) verificou que a essa relação entre os *bytes* da chave e as *sboxes* poderia ser expandida, tendo em vista que as *sboxes* do Twofish possuíam dependência de apenas 2 *bytes* da chave, o que já tornava a implementação *white-box* mais resistente ao *BGE-Attack*, aumentando a complexidade de execução desta de 2^{24} para 2^{40}, em cada um dos *rounds*; conforme apontado por Klinec et al. (2013). Contudo, mesmo com esse aumento no custo da execução dessa criptoanálise, o tempo de execução de 2^{40} ainda é computacionalmente alcançável, e pode ser paralelizado e ainda, otimizado .

Por isso, em sua nova proposta de geração de *sboxes*, Klinec et al. (2013) propôs que a *sbox* possuísse dependência em função de 13 *bytes* da chave, o que segundo ele, tornava o *BGE-Attack* bem mais custoso, sendo necessário 2^{128} passos para sua execução . Esse conceito introduzido por Klinec et al. (2013) também foi mantido por Bačinská (2015) que, implementou a *sboxgen* proposta por aquele. A função da *sboxgen* é apresentada na figura 5.2 e dentre as variáveis desta função, tem-se que:

- j - Corresponde ao índice da chave
- l - Nível de aninhamento.

- K - Corresponde a um *byte* derivado da chave, mas na notação de Bačinská (2015), esse utiliza explicitamente o *byte* k_4, provavelmente para fins de exemplificação. Enquanto Klinec et al. (2013) utiliza o K presente na posição j+1
- $q'_{j,l}$ = Esses $q'_{j,l}$'s correspondem as permutações de 8 *bits* do Twofish

$$sboxgen(j,l,x) = \begin{cases} q'_{j,0}[x] & \text{if } l = 0 \\ q'_{j,l}[sboxgen(j,l-1,x) \oplus k_{4\cdot(l-1)+j}] & \text{otherwise} \end{cases}$$

FIG. 5.2: Funcionamento da *sboxgen*. Proposta por Klinec et al. (2013). Ilustração extraída de Bačinská (2015)

Os *bytes* de chave que são utilizados na geração das *sboxes* são diferentes daqueles gerados para a encriptação do algoritmo; sendo a mesma função de expansão, mas os valores passados ao parâmetros responsáveis pela geração das chaves, ligeiramente diferentes.

$$k^r_{Sbox} = \begin{cases} hash_{par}(key||"SBOXconstant", salt) & \text{if } r = 0 \\ hash_{par}(k^{r-1}||key||"SBOXconstant", salt) & \text{otherwise} \end{cases}$$

FIG. 5.3: Expansão de chave para a *sbox*. Proposta por Klinec et al. (2013), figura de Bačinská (2015)

Percebe-se a grande semelhança existente entre as funções representadas nas figuras 5.1 e 5.3, em que na segunda, além da chave utilizada como *input*, tem-se uma concatenação dessa chave com a constante "SBOXconstant"; deste modo, as chaves geradas para as *sboxes* são distintas, o que não permite uma associação entre as duas funções e faz com que a possível descoberta das chaves de uma, não interfira no funcionamento da outra.

5.2.3 O USO DE MATRIZES MDS COM VALORES DINÂMICOS

O processo de geração das sub-chaves idealizado por Klinec et al. (2013) e implementado por Bačinská (2015) na geração da matrizes MDS é bem parecido com o das demais funções descritas anteriormente. Se diferenciando apenas pela *string* utilizada em conjunto com a chave na entrada da função *hash*; conforme demonstra a figura 5.4

Conforme demonstrado na figura 5.4, cada chave de *round* é utilizada em conjunto com a chave principal e a *string* "MDSconstant", para gerar as chaves dos demais *rounds*; o primeiro *round* é única exceção nesse funcionamento por ainda não possuir uma chave de *round*, utilizando apenas a chave principal e a *string* em sua entrada.

$$k^r_{MDS} = \begin{cases} hash_{par}(key||"MDSconstant", salt) & \text{if } r = 0 \\ hash_{par}(k^{r-1}||key||"MDSconstant", salt) & \text{otherwise} \end{cases}$$

FIG. 5.4: Expansão de chave para as matrizes MDS. Proposta por Klinec et al. (2013), figura de Bačinská (2015)

Com essas chaves que foram geradas, Bačinská (2015) apresenta no quarto capítulo de sua dissertação, um algoritmo que demonstra a utilização de cada chave, para gerar as matrizes MDS de seus respectivos *rounds*; no entanto, o funcionamento desse algoritmo, sem a observância do código fonte, é um tanto quanto obscuro e não permite inferir de forma simples o processo executado por Bačinská (2015).

Essas matrizes geradas por Bačinská (2015), foram fundamentadas no trabalho de Klinec et al. (2013) que, abordou a substituição de matrizes MDS de 4x4 *bytes*, por matrizes de 16x16 *bytes*; visando ampliar a difusão do algoritmo, Klinec et al. (2013) por sua vez, retomou o trabalho de Nakahara Jr e Abrahao (2009), que demonstrou a necessidade de se utilizar matrizes MDS maiores, para se obter uma taxa de difusão mais alta e alcançar o efeito avalanche em cada um dos *rounds*, o que não era obtido pela função *mixcolumns*.

O trabalho de Abrahao (2007), tratado de forma mais ampla no capítulo 3 dessa dissertação, serviu de base para a implementação WBAES+ de Bačinská (2015). O processo de geração das matrizes MDS executado por Bačinská (2015) é detalhado a seguir.

5.2.3.1 GERAÇÃO DAS MATRIZES MDS PARA UM ÚNICO *ROUND*

1 - Cada *byte* da chave do *round* tem os seus seis primeiros e seis últimos blocos armazenados em um *Set*, que possui o comportamento semelhante ao de um vetor, mas que não armazena valores duplicados. Esse processo é apresentado na figura 5.5.
2 - Os primeiros 14 *bytes* da primeira linha da matriz, são os primeiros 14 valores do *Set*; conforme demonstra a figura 5.6 e o algoritmo 3. Caso o *Set* possua menos de 14 valores, a primeira linha não poderá ser preenchida por completo e a matriz MDS não será gerada em função da chave; para esse caso, será usada a matriz estática apresentada por Nakahara Jr e Abrahao (2009).

3 - O 15º *byte* da primeira linha é escolhido a partir dos outros 14 *bytes* anteriores, sendo necessário realizar um XOR entre esses 14 *bytes*, o 15º *byte* a ser selecionado e o número 1. Neste caso, o resultado deve ser um valor diferente de todos esses 15 *bytes*, a fim de garantir que todos os valores sejam diferentes e ainda sim atendam a propriedade

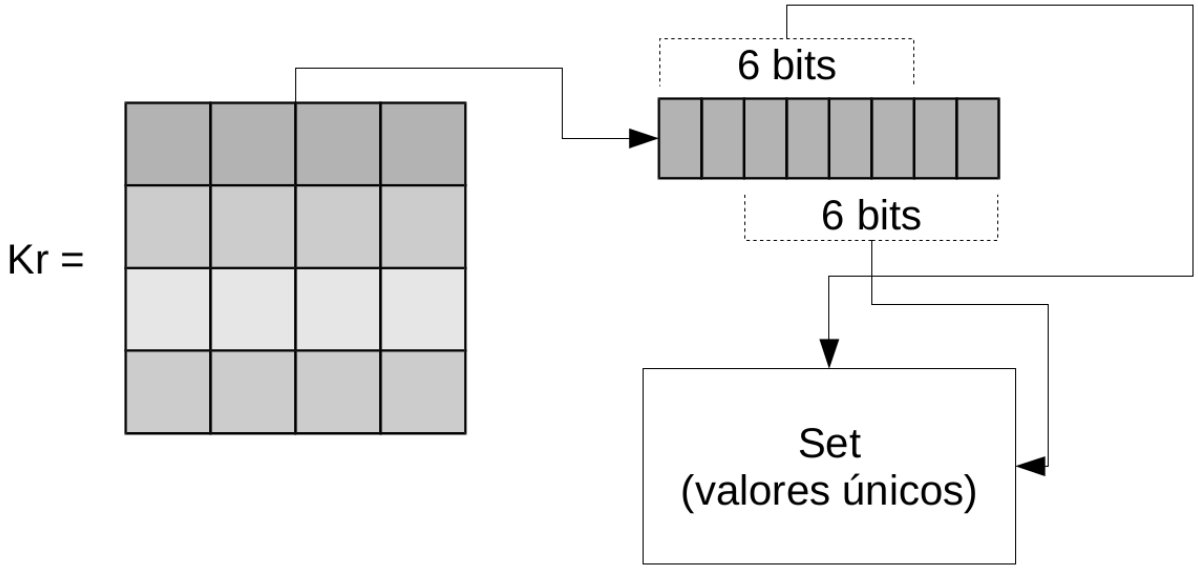

FIG. 5.5: Os seis primeiros e os seis últimos *bits* de cada *byte* de K_r são armazenados em um *Set*, que armazena apenas valores únicos.

primeiraLinha

Set [0]	Set [1]	Set [2]	Set [3]	Set [4]	Set [5]	Set [6]	Set [7]	Set [8]	Set [9]	Set [10]	Set [11]	Set [12]	Set [13]		

FIG. 5.6: Os primeiros 14 blocos de 6 *bits* do *Set*, são os primeiros 14 *bytes* da primeira linha.

da Matriz Cauchy descrita por Bačinská (2015), em que um XOR realizado entre todos os seus *bytes* deve resultar no valor 1. O algoritmo 3 demonstra o processo para selecionar o 15º *byte* da primeira linha.

Algoritmo 3: PREENCHE PRIMEIRALINHA

Entrada: $Set, primeiraLinha$

```
1  início
2      PL = primeiraLinha
3      para i = 0 até Set.length faça
4          se i < 14 então
5              PL_[i] = Set_[i]
6              i = i + 1
7          fim
8          senão
9              se PL.contem(PL_[0] ⊕ PL_[1] ⊕ ··· ⊕ PL_[13] ⊕ Set_[i] ⊕ 1) então
10                 i = i + 1
11             fim
12             senão
13                 PL_[14] = Set_[i]
14                 PL_[15] = PL_[0] ⊕ PL_[1] ⊕ ··· ⊕ PL_[14] ⊕ 1
15                 i = Set.length
16             fim
17         fim
18     fim
19 fim
```

Após a execução do algoritmo 3, responsável por preencher o vetor *primeiraLinha*, é verificado se este possui todas as suas 16 posições preenchidas. Em caso positivo, as demais linhas serão permutações predefinidas deste, até que se tenha como resultado uma matriz 16x16.

Caso o vetor *primeiraLinha* não possua todas as suas posições preenchidas, será utilizada a matriz estática construída por Abrahao (2007) em sua dissertação e por Nakahara Jr e Abrahao (2009) em seu artigo. Essa matriz também é apresentada no capitulo 3, que trata sobre a difusão.

6 RECOMENDAÇÕES *WHITE-BOX* E SUAS APLICAÇÕES PARA A CRIAÇÃO DO WBTWOFISH E WBTWOFISH+

Os capítulos anteriores desta dissertação, se dedicaram principalmente, a descrever os conceitos necessários para que um algoritmo em *white-box* não permita à obtenção de sua chave criptográfica de forma simples, a partir da supervisão de um atacante nos processos de encriptação e decriptação.

A fim de garantir que tanto a encriptação quando a decriptação apresentassem o mesmo grau de difusão, foi sugerido a utilização de matrizes MDS involuntórias, para aqueles algoritmos que utilizam esse tipo de matriz, como é o caso do Twofish. No entanto, antes que esse conceito das matrizes MDS fosse apresentado, alguns assuntos mais preliminares, que surgiram na proposta da implementação *white-box* de Chow et al. (2003), foram introduzidos.

O capítulo 2 tratou daquilo que foi abordado por Chow et al. (2003) a nível de estrutura das funções, englobando tópicos como as tabelas de pesquisa, que demonstram como algumas pequenas modificações na estrutura das operações do algoritmo, podem inserir obstáculos entre a memória e a chave principal, que é a principal informação pretendida pelo atacante.

Dentre as alterações iniciais, tem-se como a mais explícita, a reordenação das funções *AddRoundKey* e *ShiftRows*, que permitiu a K_0 não ser mais armazenada diretamente na memória; além dessa alteração, este capítulo terá como objetivo, discutir também tudo aquilo que foi apresentado ao longo desta dissertação; para permitir que os conceitos que foram tratados, sejam aderentes ou adaptáveis à qualquer algoritmo de cifra simétrica, planejado para um contexto *white-box*.

No transcorrer deste trabalho, foi identificado que os algoritmos de criptografia simétrica compartilham algumas características que podem ser enumeradas e tratadas de forma individual. A partir desse tratamento, pode-se descrever no que consiste cada uma dessas características e destacar os pontos que nelas foram aprimorados, no processo de transformação do AES até o WBAES+, para definir quais alterações podem tornar um algoritmo de criptografia simétrica mais adequado ao contexto *white-box*.

As características mais comuns compartilhadas entre esse conjunto de algoritmos, são listadas a seguir e, posteriormente, algumas delas são detalhadas nas próximas seções.

a) Os algoritmos de criptografia simétrica possuem funções não-lineares e funções lineares

b) Esses algoritmos fazem uso de uma chave para cifrar e decifrar um conteúdo; essa chave é expandida de forma que possa ser utilizada em outros *rounds* ou funções.

c) As funções desses algoritmos recebem x *bits* e retornam y *bits*, podendo x ser igual a y, ou não.

d) Os possíveis valores de entrada e saída de uma função estão limitados a sua quantidade de *bits*, a menos que se faça uso de segurança local.

6.1 FUNÇÕES NÃO-LINEARES

Sendo essas funções responsáveis por introduzirem confusão, o objetivo é obter em um *white-box*, o mesmo grau de confusão alcançado em *black-box*. Em geral, as funções não-lineares são representadas por funções de substituição.

Analisando essas funções, é possível notar que se há uma substituição de valores, haverá também uma ou várias tabelas que determinam os valores dessa substituição e, frequentemente, os valores dessas tabelas de substituição são estáticos e publicamente conhecidos; como é o caso dos algoritmos AES e DES, por exemplo.

Em um ambiente *black-box*, uma tabela de substituição com valores estáticos não gera de forma alguma as mesma debilidades que em um ambiente *white-box*. Sendo a *sbox* do AES, por exemplo, mesmo que estática em um ambiente *black-box*, suficiente para se alcançar um grau de segurança que não pode ser obtido considerando-se o mesmo algoritmo em um ambiente vulnerável.

Isso se deve ao fato da relação existente entre a chave e o criptograma em um ambiente *black-box*, em que tanto a chave quanto o texto claro passaram por todos os *rounds* do algoritmo até resultar no criptograma, ser bem mais complexa do que se comparada ao ambiente *white-box*, em que o atacante tem acesso a criptogramas parciais que correspondem a saída de cada um dos *rounds*.

No cenário *white-box*, em que o texto claro é conhecido, a única incógnita para o atacante é a chave criptográfica; no entanto, como os valores da matriz ou tabela de substituição são publicamente conhecidos, o que é explorado como fraqueza por Billet et al. (2005) e, pelo fato de o atacante obter o criptograma ao final de cada round, que é quando fatalmente ocorre a persistência dos dados na memória, verifica-se que as características do algoritmo necessitam de fato serem modificadas.

O princípio de Kerckhoffs diz que a segurança do sistema deve estar garantida mesmo que todo o sistema, exceto a chave, seja de conhecimento público, (KERCKHOFFS, 1883). Desta forma, uma cifra deve permanecer segura mesmo que um atacante conheça todos os detalhes do algoritmo de criptografia empregado.

Visando centrar a segurança ainda mais na chave criptográfica e fortalecer o algoritmo contra o BGE-Attack, pode-se tornar os valores da matriz de substituição dependentes da chave criptográfica; assim, será possível determinar a geração da matriz baseado no único valor desconhecido pelo atacante, que é a chave.

No início desta análise, o AES e DES foram citados como exemplo de algoritmos que possuem matrizes de substituição com valores estáticos e, em função desta característica, foi proposto por Klinec et al. (2013) e implementado por Bačinská (2015), a utilização de algumas funções do Twofish para reduzir essa debilidade do AES. O problema da matriz com valores estáticos foi solucionado adotando-se, inicialmente, a mesma *sbox* utilizada pelo Twofish, que apresenta valores dependentes da chave e, posteriormente, expandido-se essa dependência, que era em função de 2 *bytes* da chave em cada *sbox*, para 13 *bytes* da chave; como foi mencionado no capítulo 4, o que permite, segundo Klinec et al. (2013), aumentar o custo computacional do BGE-Attack para 2^{128}.

6.2 FUNÇÕES LINEARES

As funções lineares apresentam um objetivo diferente das não lineares, tendo como intuito introduzir difusão ao algoritmo e dissipar os padrões existentes entre a entrada, texto claro, e a saída do algoritmo, criptograma.

Ao introduzir difusão no algoritmo, pretende-se que os padrões existentes no texto claro não sejam transportados para o texto cifrado e, que a mudança de qualquer *bit* no texto claro cause a mudança de muitos ou todos os *bits* no criptograma, obtendo-se o resultado conhecido como efeito avalanche. Isso é alcançado pela maioria dos algoritmos conhecidos, que são executados em um contexto *black-box*, pelo fato do atacante só ter acesso ao criptograma quando este já sofreu todas as alterações previstas pelas funções do algoritmo, sendo efeito avalanche de fato alcançado.

No entanto, em um contexto *white-box*, considerando algoritmos como o AES ou Twofish, por exemplo, tem-se que o acesso de um atacante ao criptograma ocorre ao final de um único *round*, fazendo com que a difusão não seja completa e o efeito avalanche não seja obtido.

Sobre as **funções não lineares**, foi exemplificado que, em geral, essas são represen-

tadas por **funções de substituição**; no entanto, quando o assunto é **funções lineares**, trata-se, normalmente, de **operações de permutação** e/ou daquelas operações que atuem sobre corpos finitos e fazem uso das **matrizes MDS** durante o processo de cálculo.

Tanto o AES, quanto o WBAES de Chow et al. (2003), utilizam matrizes MDS de 128 *bits*. Essas matrizes, em um contexto *white-box*, não alcançam o efeito avalanche ao final de um único *round*, que é quando o atacante fatalmente tem acesso a informação persistida na memória.

A fim de resolver esse problema, uma das soluções que podem ser adotadas por todos os algoritmos que fazem uso de uma matriz semelhante a do *mixcolums*, como é caso do Twofish, é a expansão da matriz de acordo com o tamanho do *state*. Sendo o *state* igual a 128 *bits*, o ideal é que a matriz MDS possua o quadrado de *bits* do *state*, totalizando 128^2 *bits*, o que resulta em uma matriz 16x16 *bytes* para oferecer difusão completa e alcançar o efeito avalanche em um único round, conforme mostra a figura 6.1.

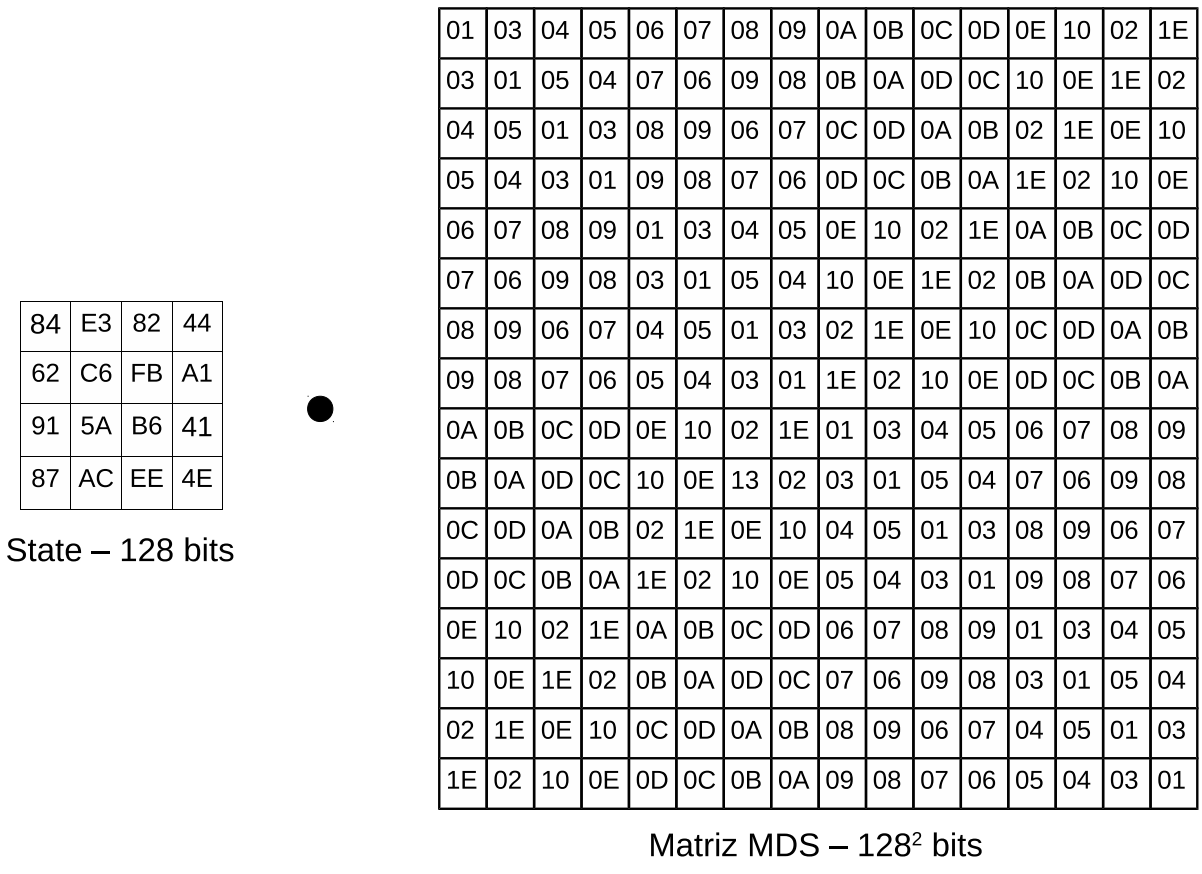

FIG. 6.1: Ilustração de uma forma de se alcançar o efeito avalanche em um único *round*, através de uma relação em que a matriz MDS possui, pelo menos, o quadrado de *bits* do *state*

Além de não alcançarem o efeito avalanche em um único round, tanto a matriz uti-

lizada pelo AES, quanto a utilizada pelo Twofish, fazem uso de valores estáticos. Uma forma de tornar esses valores dinâmicos e dependentes de chave, é fazer uso dos conceitos apresentados no capítulo 5, em que é exemplificado o processo de geração das Matrizes Cauchy, adotadas pelo WBAES+.

6.3 UTILIZANDO FUNÇÕES *ONE-WAY HASH* NA MANIPULAÇÃO DA CHAVE CRIPTOGRÁFICA

No capítulo 5, houve uma abordagem específica à chave criptográfica, em que foi tratada a importância da utilização de funções *one-way hash* como KDF, a fim de não permitir que a chaves expandidas fossem um ponto de partida para a chave principal.

Esta seção visa apenas reforçar a importância do uso de uma função *hash* nos processos que envolvam a chave criptográfica em um contexto *white-box*. Não há como correr riscos de que a chave principal possa ser deduzida a partir de suas variantes. Deste modo, a maneira mais simples de garantir que isto não aconteça, é através de uma função *one-way hash* na geração das chaves derivadas. A utilização de um tipo de *hash* como o *scrypt*, por exemplo, é ainda mais interessante, tendo em vista que este utiliza técnicas de *memory-hard* que objetivam inviabilizar ataques de força bruta, conforme foi apresentado no capítulo 5.

6.4 A ESCOLHA DE UM ALGORITMO COMO ESTUDO DE CASO

Ao longo desta dissertação foi realizado um levantamento que incluiu o início da implementação *white-box*, realizado por Chow et al. (2003), até a implementação WBAES+ de Bačinská (2015). Através dessas implementações, foi possível realizar uma análise dos métodos aplicados ao algoritmo AES, para que os conceitos na aplicação desses métodos pudessem ser aplicados em outros algoritmos; tendo em vista que as cifras simétricas compartilham diversos pontos em comum.

Com o objetivo de não só generalizar os conceitos aplicados ao AES, mas também permitir a projeção desses conceitos em outra cifra, optou-se por utilizar o Twofish como estudo de caso. Esse algoritmo, assim como o Rijndael, foi um dos 5 finalistas do concurso AES; além disto, o Twofish também teve a sua *sbox* utilizada como referência pelo WBAES+, o que permite a ele, uma posição de destaque dentre os possíveis candidatos para a aplicação desses conceitos.

6.5 O TWOFISH E SUA ESTRUTURA

O Twofish é uma cifra de blocos de 128 *bits* que aceita chaves de 128, 192 e 256 *bits*. A cifra é uma rede de *Feistel* de 16 *rounds* com uma função bijetora F, contendo 4 *sboxes* de 8x8 *bits*, com dependência de chave; Uma matriz MDS fixa de 4x4 *bytes* sobre GF(2^8); Uma transformação *pseudo-Hadamard*; Rotações *bit* a *bit* e uma expansão de chave. Além disto, o Twofish também possui uma operação de *Whitening* em sua entrada e outra em sua saída.

A estrutura do Twofish é representada pela figura 6.2, extraída de Schneier et al. (1998).

6.6 UMA IMPLEMENTAÇÃO *WHITE-BOX* PARA O TWOFISH

No capítulo 2, que tratou sobre as tabelas de pesquisa, verificou-se que a restruturação das funções do AES se deveu ao fato de sua estrutura original, quando exposta a um ambiente *white-box*, não executar muitas transformações na chave, antes que esta fosse armazenada na memória, fazendo com que ela fosse facilmente resgatada por um atacante.

6.6.1 AS IMPLICAÇÕES DAS TÉCNICAS DE *INPUT* E *OUTPUT WHITENING* EM UM CONTEXTO *WHITE-BOX*

No AES, a função *AddRoundKey* armazenava a chave diretamente na memória, tendo realizado apenas um XOR entre esta e o texto claro. Esse processo, também ocorre no Twofish que, faz uso das técnicas de *input* e *output Whitening*; essa técnica de *Whitening* consite em realizar um XOR entre o *state* e a chave, alterando assim o *state* antes que ele seja introduzido nas funções do algoritmo ou se transforme definitivamente no criptograma.

Em um contexto *black-box* não há as mesmas implicações negativas de uma técnica de *Whitening* do que, quando esta é realizada em um ambiente *white-box*. Em um ambiente vulnerável, a técnica terá o mesmo resultado produzido pelo *AddRoundKey* na estrutura original do AES, contudo, no Twofish a técnica de *Whitening* não é uma função projetada em todos os *rounds*, como é o *AddRoundKey* no AES. Isto implica em algumas mudanças no Twofish para um ambiente *white-box*, que são um pouco mais severas do que se comparadas as existentes entre o WBAES e o AES.

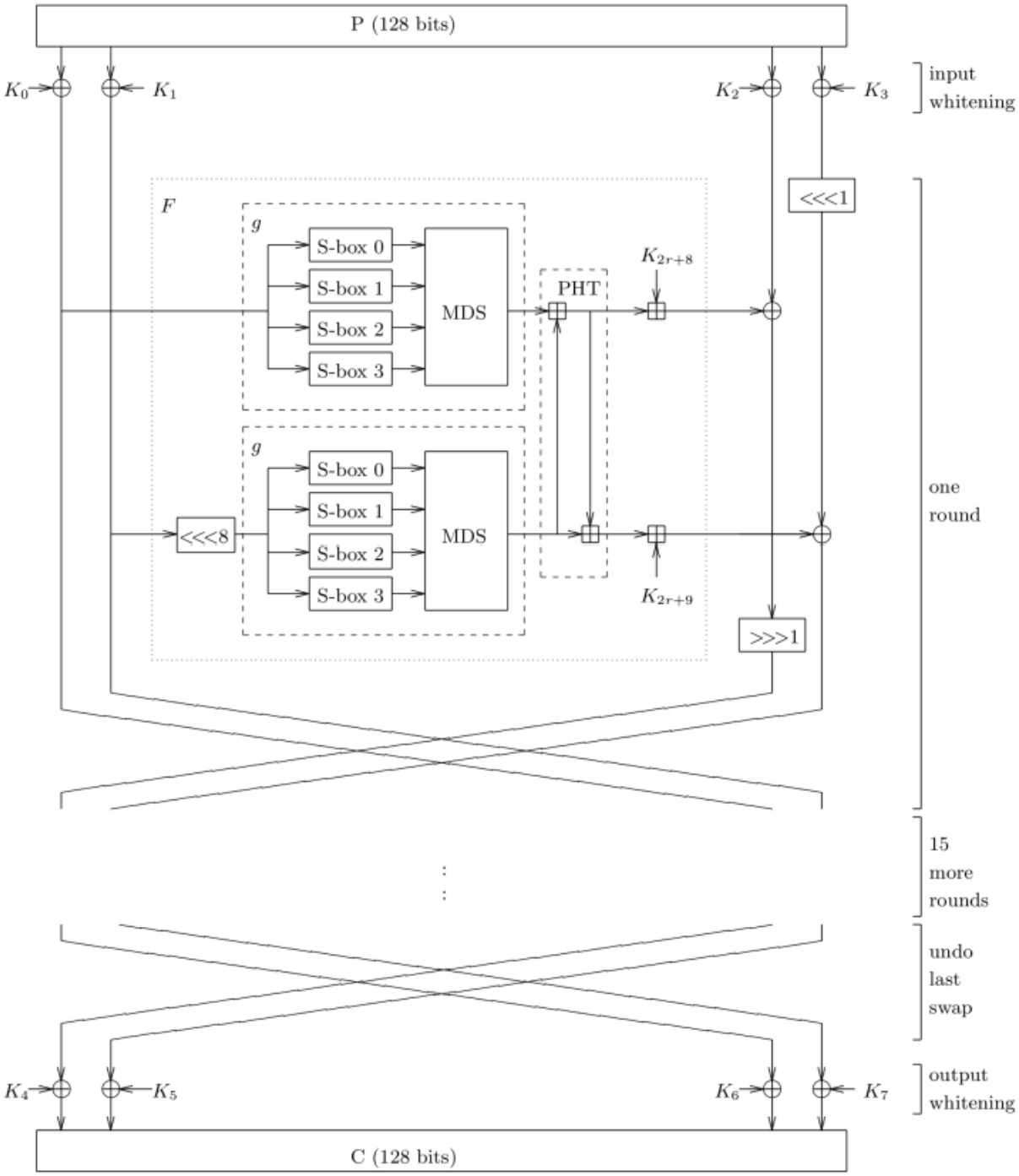

FIG. 6.2: Twofish projetado por Schneier et al. (1998).

6.6.2 A CRIAÇÃO DO WBTWOFISH: UMA VERSÃO DO TWOFISH QUE NÃO ALTERA CRIPTOGRAMA

Para manter o mesmo criptograma gerado pelo Twofish, é necessário preservar algumas das características do algoritmo, como é o caso da técnica de *Whitening* que armazena na memória apenas um XOR entre a chave criptográfica e o *state*, permitindo a um atacante

inferir a chave facilmente utilizando alguma técnica como a demonstrada por Kerins e Kursawe (2006), caso o seu funcionamento não seja adaptado ao contexto *white-box*

Sendo necessário utilizar o funcionamento apresentado pelo *Whitening* e, para oferecer um maior grau de dificuldade na recuperação da chave, foram efetuadas algumas transformações no modo como essas funções estavam organizadas, gerando um resultado similar ao apresentado por Chow et al. (2003) quando as suas tabelas de pesquisa foram introduzidas no WBAES.

A concepção de Chow et al. (2003) com suas tabelas era de que fosse possível minimizar as informações que "vazavam" para a memória e, permitir a utilização de codificações e das bijeções misturadoras; a primeira restruturação de Chow et al. (2003) foi trazer a operação de *AddRoundKey*, que realizava um XOR entre o *state* e k_0, a chave principal, para dentro da operação *for*. Através disso, como é demonstrado no capítulo 2, Chow et al. (2003) fizeram com que em cada *round* a chave criptográfica fosse apenas armazenada após ser transformada em conjunto com o *state* pelas demais funções, como *Mixcolumns* e *Shiftrows*; sendo necessário ainda desfazer as codificações e as bijeções.

6.6.2.1 ALTERANDO A ORGANIZAÇÃO DOS ROUNDS DO WBTWOFISH

Para o Twofish, o comportamento adotado deve ser um tanto diferente naquilo que tange a reestruturação das funções. No AES, a operação *AddRoundKey* é recorrente em cada *round*, o que permitiu que esta fosse interiorizada na operação *for* de maneira natural. Contudo, tanto a operação de *input* quanto a de *output Whitening* não ocorrem em cada um dos *rounds* do Twofish, o que não permite que essas operações sejam incorporadas aos *rounds* de forma simples, como foi efetuado com o WBAES.

Assim, o processo efetuado nesta dissertação executa uma proposta inversa, fazendo com que a estrutura interna dos *rounds* seja adequada às operações de *Whitening*, que ocorrem antes e depois desses *rounds*. O Twofish originalmente possui 16 *rounds*, nesta implementação do WBTwofish, as operações referentes aos dois primeiros *rounds* são retiradas da estrutura de repetição *for* e colocadas antes de sua ocorrência; o mesmo ocorre para as operações referentes aos dois últimos *rounds* do Twofish que, também são extraídas da operação *for*, sendo executadas depois de sua ocorrência.

Uma exemplificação desse novo comportamento é apresentada na figura 6.3 e altera o fluxo de execução original do Twofish, apresentado na figura 6.2.

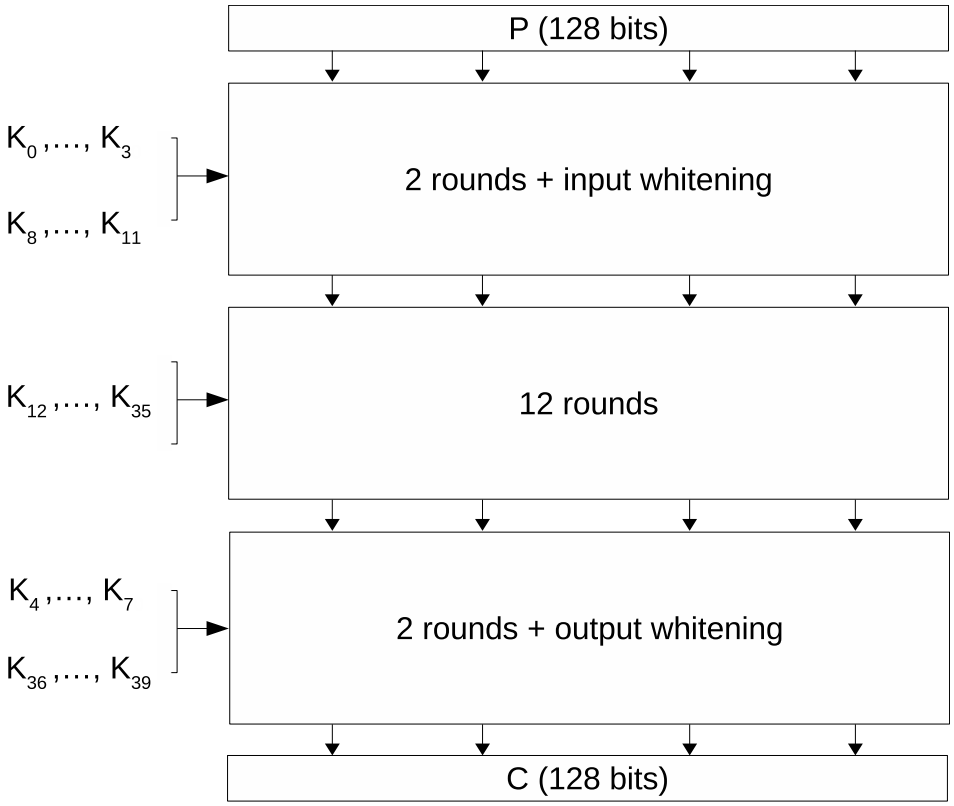

FIG. 6.3: WBTwofish: Alteração na distribuição dos *rounds* do Twofish.

6.6.2.2 A ESTRUTURA DO WBTWOFISH DE ACORDO COM A REDE DE FEISTEL

Na estrutura original do Twofish, o texto claro é dividido em quatro partes de 32 *bits* que realizam um XOR com quatro palavras de chave do mesmo tamanho, K_0, K_1, K_2 e K_3, respectivamente. O resultado dessas operações consiste em R_0, R_1, R_2 e R_3; conforme demonstra a figura 6.2. R_0 e R_1 são introduzidos na função F, sendo a saída de R_0, combinada com R_2; através de uma operação XOR seguida por uma permutação de 1 *bit* à direita, resultando em C_2.

Já a saída de R_1, é combinada através de uma operação XOR com R_3, após R_3 sofrer uma permutação de 1 *bit* à esquerda, resultando em C_3. Em seguida, C_2 e C_3 trocam de lugar com R_0 e R_1 e são introduzidos no próximo *round*. Esse comportamento, segue o proposto pela rede de Feistel, que é um método geral de transformação de qualquer função(Normalmente chamada de função F) em uma permutação. Tendo sido inventado por Feistel et al. (1975) e implicando no comportamento demonstrado na figura 6.2, ou de maneira mais específica, na figura 6.4.

Como é apresentado em (SCHNEIER et al., 1998) e (SCHNEIER; KELSEY, 1996), dois *rounds* de uma rede de Feistel constituem o chamado "ciclo". Em um ciclo, cada *bit* do

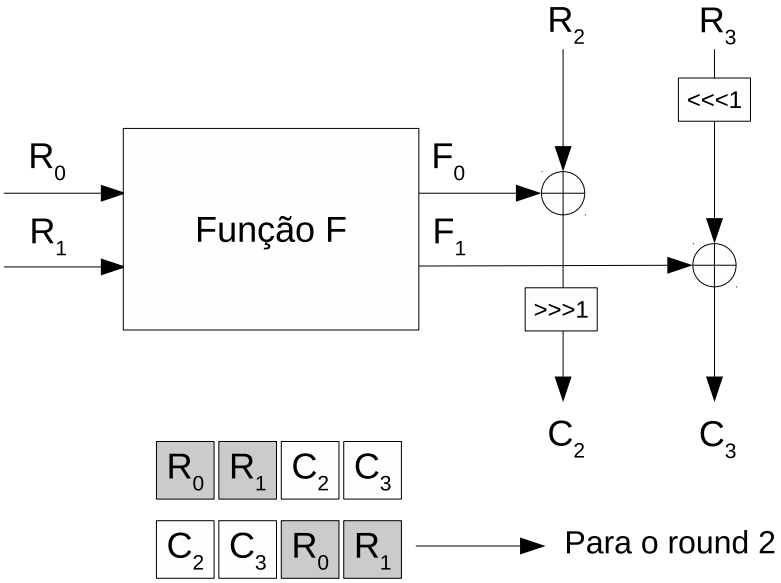

FIG. 6.4: Simplificação do funcionamento do primeiro *round* do Twofish implementando a rede de Feistel.

bloco de texto, nesse caso o *state*, representado por R_0, R_1, R_2 e R_3, é modificado pelo menos uma vez.

Por isso, foi necessário destacar dois *rounds* para cada uma das operações de *Whitening*, caso contrário, os *bytes* combinados com a chave não teriam sido completamente alterados antes de serem persistidos na memória.

6.6.2.3 MODIFICANDO O FUNCIONAMENTO DOS DOIS PRIMEIROS *ROUNDS* PARA MINIMIZAR O VAZAMENTO DA CHAVE

Através dos dois primeiros *rounds* que, foram destacados dos demais para atuarem em conjunto com a operação de *input Whitening*, foram definidas quatro variáveis correspondentes as saídas modificadas das palavras de 32 *bits*: C_0, C_1, C_2 e C_3; essas variáveis são resultado das transformações ocorrentes na função F e das permutações da rede de Feistel. Conforme mencionado anteriormente, após dois *rounds*, todas as palavras de 32 *bits*, já teriam os seus *bits* modificados pelo menos uma vez.

No capítulo 2, que tratou das tabelas de pesquisas, foi demonstrado a importância de se aglutinar funções e de fazer com que a chave fosse acrescentada na memória apenas quando esta sofresse diversas alterações previstas pelas funções do algoritmo.

Na implementação convencional do Twofish, a operação XOR que ocorre entre o *state* e a chave é executada antes que haja qualquer outra operação do algoritmo; contudo, nesta implementação do WBTwofish, esse comportamento precisou ser levemente alterado.

A primeira alteração foi fazer com que o XOR entre as duas primeira palavras de 32 *bits* da chave, K_0 e K_1, ocorressem diretamente na entrada da *sboxes*, não gerando uma variável adicional para ser persistida na memória.

Dentro da função F, existem duas funções g, que na notação abaixo serão representados por função g_0 e g_1. A descrição dessas funções é apresentada a seguir.

$$g_0 = MDS(Sboxes(P_0 \oplus K_0))$$
$$g_1 = MDS(Sboxes((P_1 \oplus K_1) \lll 8))$$

As saídas das funções g são combinadas entre si, através de uma transformação *Pseudo-Hadamard*(PHT) e, em seguida, essas saídas são somadas com as saídas das funções h, como demonstra a figura 6.5.

As funções h são responsáveis pela expansão de chave, tendo como resultado as chaves K_{2r+8} em F_0 e K_{2r+9} em F_1. Já a transformação *Pseudo-Hadamard* é uma operação simples de mistura que, dada duas entradas, a e b, realiza as operações demonstradas a seguir.

$$\text{a'} = \text{a} + \text{b} \bmod 2^{32}$$
$$\text{b'} = \text{a} + 2\text{b} \bmod 2^{32}$$

Na implementação convencional do Twofish, para se obter o resultado de F_0 e F_1, que é apresentado na figura 6.5, é utilizada a operação PHT descrita na anteriormente; contudo, a notação presente em Schneier et al. (1998) apresenta mais variáveis do que a demonstrada abaixo na implementação do WBTwofish, em que o objetivo e aglutinar tantas informações quanto possível. Sendo assim, F_0 e F_1 são descritos das seguinte forma:

$$F_0 = (MDS(Sboxes(P_0 \oplus K_0)) + MDS(Sboxes((P_1 \oplus K_1) \lll 8)) + K_{2r+8}) \bmod 2^{32}$$
$$F_1 = (MDS(Sboxes(P_0 \oplus K_0)) + 2{*}(MDS(Sboxes((P_1 \oplus K_1) \lll 8))) + K_{2r+9}) \bmod 2^{32}$$

No entanto, apesar das chaves K_0 e K_1 não serem mais armazenadas nem em R_0, nem em R_1, para o *round* 1; ainda é necessário fazer com que todas as transformações cabíveis sejam aplicadas, antes que essas variáveis sejam transferidas para o próximo *round*. Para isso, pode-se fazer também com que tanto R_2 quanto R_3 incorporem a aglutinação que interiorizou K_0 e K_1. Ao final desses dois rounds, agora aglutinados, haverão quatro variáveis: C_0, C_1, C_2 e C_3, que representarão o encadeamento máximo das operações.

Tanto K_2 quanto P_2, que resultavam em R_2; agora são imediatamente conectadas com as operações que seriam armazenadas em F_0; tornando-se C_2, com mais aglutinações:

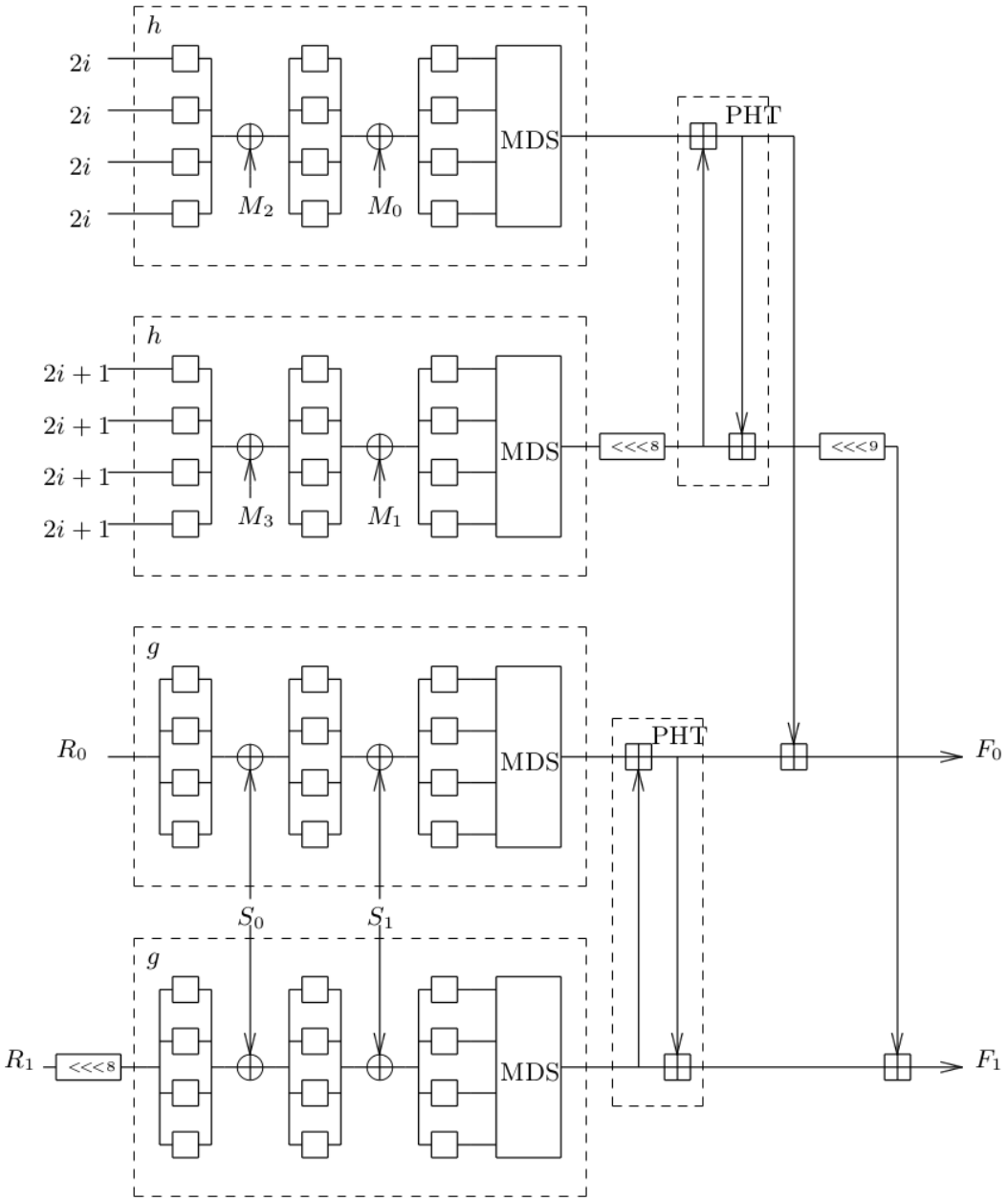

FIG. 6.5: Uma visão de um único round da função F (chave de 128 *bits*), extraído de Schneier et al. (1998).

$$C_2 = ((MDS(Sboxes(P_0 \oplus K_0)) + MDS(Sboxes((P_1 \oplus K_1) \lll 8)) + K_{2r+8} \oplus P_2 \oplus K_2) \bmod 2^{32}) \oplus \psi$$

Além de K_2 e P_2, em C_2 também é realizado uma operação XOR com ψ, que nesse caso, representa um aleatório qualquer de 32 *bits*. Como a permutação dos *bits* de C_2 ocorre após a realização de um XOR com o que era o resultado de F_0 na implementação convencional do Twofish; será necessário que essa permutação seja realizada sobre C_2, somente quando este receber todos as sua devidas variáveis. Dessa forma, quando a permutação dos *bits*

for realizada, pode-se remover o aleatório inserido, através da operação XOR. O uso desse aleatório visa aumentar a segurança local, como foi tratado no capítulo 3, dentro da seção que abordou as codificações. Sendo assim, ao sofrer a permutação prevista, C_2 será equivalente à:

$$C_2 = (C_2 \oplus \psi) \ggg 1$$

Se tratando de C_3, este pode assumir um comportamento diferente de C_2. Como na implementação convencional do Twofish, R_3 é permutado antes de realizar um XOR com a saída das função F_0, pode-se atribuir todas as variáveis de uma vez em C_3, incluindo a permutação; conforme é mostrado a seguir:

$$C_3 = (MDS(Sboxes(P_0 \oplus K_0)) + 2^*(MDS(Sboxes((P_1 \oplus K_1) \lll 8))) + K_{2r+9} \oplus ((P_3 \oplus K_3) \lll 1)) \bmod 2^{32}$$

Já C_0 e C_1, transformados no *round* 2 e correspondentes a R_0 e R_1, que sofrem as mesmas alterações que resultaram em C_2 e C_3, mas com valores diferentes; tem-se que o processo de geração dessas variáveis será bastante semelhante ao realizado em C_2 e C_3. C_0 é composto da seguinte maneira:

$$C_0 = ((MDS(Sboxes(C_2)) + MDS(Sboxes(C_3 \lll 8)) + K_{2r+8} \oplus P_0 \oplus K_0) \bmod 2^{32}) \oplus \psi$$

C_0 executa um processo parecido com o realizado por C_2, com a diferença de que as variáveis inseridas nas *sboxes* são C_2 e C_3, respectivamente. Além disso, tanto P quanto as chaves, são diferentes daquelas utilizadas por C_2 e, sendo o *round* 2, a chave correspondente é K_{2r+8}. É importante ressaltar, que assim como C_2, C_0 também realiza uma operação XOR com uma palavra de 32 *bits*, aleatória, que aqui é representada por Φ.

$$C_0 = (C_0 \oplus \Phi) \ggg 1$$

Acima, foi mostrado C_0, após sofrer a permutação prevista pela rede de Feistel, antes que fosse introduzido no *round* seguinte.

Por último, tem-se C_1 que segue um comportamento semelhante ao realizado por C_3 no round anterior; não sendo necessário, pelos mesmos motivos, realizar a permutação em uma etapa separada. C_1 é apresentado por:

$$C_1 = (MDS(Sboxes(C_2)) + 2^*(MDS(Sboxes(C_3 \lll 8))) + K_{2r+9} \oplus ((P_1 \oplus K_1) \lll 1)) \bmod 2^{32}$$

6.6.3 O FUNCIONAMENTO DO WBTWOFISH PARA OS DEMAIS *ROUNDS* E O SEU COMPORTAMENTO DURANTE A DECRIPTAÇÃO

O processo nos demais *rounds* é análogo ao apresentado na subseção anterior, com uma pequena diferença de que as palavras de chave utilizadas no *input Whitening*, já sofreram diversas transformações.

Os dois últimos *rounds* executam um processo ainda mais parecido com os dois primeiros, contudo, ao invés de considerar as palavras de chave do *input Whitening*, K_0, K_1, K_2 e K_3, são consideradas K_4, K_5, K_6 e K_7; e as operações de *output Whitening* são executadas sobre o *state* que, não é igual ao texto claro introduzido inicialmente no algoritmo.

Já durante o processo de decriptação, é invertido apenas o comportamento, sendo executada a operação de *output Whitening* no lugar da de *input* e vice-versa.

6.6.4 WBTWOFISH: MAIS SEGURO PARA O AMBIENTE *WHITE-BOX*

Tanto a implementação do WBTwofish, realizada em Java por este trabalho, quanto a sua descrição, efetuada ao longo desta seção, demonstram alguns aspectos utilizados por Chow et al. (2003) em sua implementação do WBAES.

Assim como na implementação de Chow et al. (2003), o WBTwofish gera o mesmo criptograma que a sua implementação original para um ambiente *black-box*, implementando alguns dos conceitos apresentados por Chow et al. (2003), como as tabelas de pesquisa e a segurança local. Além disso, em conjunto com as codificações e bijeções misturadoras, originadas no WBAES, pode-se obter uma resistência ao BGE-Attack de 2^{40} passos, necessários para se deduzir cada uma das chaves de *round* do WBTwofish. Isso é permitido devido a construção da própria *sbox* do Twofish, que faz com que o atacante tenha que deduzir dois *bytes* da chave para ter o mesmo cenário apresentado pela *sbox* do AES, conforme é demonstrado por Klinec et al. (2013). Contudo, após a implementação do WBAES+ de Bačinská (2015), verifica-se que algumas técnicas que apresentam uma maior segurança, acabam por romper a compatibilidade existente entre o criptograma do algoritmo implementado para *white-box* e sua versão anterior, prevista apenas para o contexto *black-box*.

Do mesmo modo realizado por Bačinská (2015), que gerou o WBAES+, uma versão do Twofish que implementasse alguns dos conceitos utilizados nessa implementação, como: expansão de chave com *hash*, usos de funções *memory-hard*, matrizes MDS, que apresentassem total difusão para um único *round* e, o uso de *sboxes* dependentes de vários *bytes*

da chave, poderiam contribuir para uma implementação mais resistente ao *BGE-Attack*.

A próxima seção apresenta o WBTWofish+ e especifica as técnicas que foram utilizadas ao longo de sua implementação, estendendo o WBTwofish e gerando um criptograma diferente do gerado pelo Twofish.

6.7 WBTWOFISH+: UMA NOVA VERSÃO DO TWOFISH PARA O CONTEXTO *WHITE-BOX*

No início deste capítulo, foram tratadas algumas técnicas recomendáveis para o contexto *white-box*, sendo algumas delas mais impactantes para a estrutura original do algoritmo do que aquelas utilizadas pelo WBTwofish, exemplificado na seção anterior.

Contudo, o objetivo do WBTwofish era preservar o mesmo criptograma da implementação original do Twofish; esse objetivo não foi o mesmo estipulado para o WBTwofish+. Foram aplicadas algumas das principais técnicas utilizadas no WBAES+, tendo sido algumas delas incorporadas à implementação em Java do WBTwofish+; realizada nesta dissertação.

Dentre as mudanças presentes no WBTWofish+ que estenderam o funcionamento do WBTWofish, pode-se listar as seguintes:

- Aumento no número de *bytes* da chave envolvidos em cada uma das *sboxes*. Visando prevenir a inferência dos valores da *sbox* pelo BGE-Attack.
- Utilização de uma função *hash* para expandir a chave do algoritmo e para gerar as chaves utilizadas pelas *sboxes*.

O Scrypt é a função *hash* adotada tanto para as *sboxes*, quanto para a expansão de chave. Conforme apresentado no capítulo 5, ele faz uso de *memory-hard*, o que inviabiliza ataques de força bruta. A *string* utilizada como *salt* na expansão de chave, que substitui a função h do Twofish, é a "WBTWofishPlusalt"; já para a *sbox*, a *string* utilizada é a "WBTwofish+SB".

Pelo fato da *sbox* utilizada no WBAES+ ter sido inspirada na do Twofish e, pelo fato de o funcionamento dessa *sbox* ter sido aprimorado, foram realizados alguns ajustes para que a implementação dessa função fosse adaptada ao WBTwofish+ que, assim como o WBAES+, oferece agora, uma resistência de 2^{128} passos para que os 13 *bytes* da chave sejam inferidos, inviabilizando a criptoanálise do BGE-Attack.

Além disso, o projeto do WBTwofish+, apesar de não implementar as matrizes MDS de 16x16 *bytes* em seu código, recomenda a sua utilização de forma semelhante aquela

apresentada no capítulo 4, que tratou sobre a difusão, e no capítulo 5, que abordou a dependência de chave.

6.8 EXPERIMENTOS E RESULTADOS

Ao longo desta dissertação, foram analisadas as estruturas tanto do WBAES quanto do WBAES+. Isso permitiu, tratar de forma detalhada alguns conceitos e funções importantes para algoritmos simétricos em *white-box*. A partir da aplicação de parte das técnicas analisadas no WBAES sobre o Twofish, gerou-se o WBTwofish; uma implementação *white-box* do Twofish que não altera o criptograma do algoritmo, se utilizadas a mesma chave e texto de entrada da cifra original. Além disso, é introduzido também o WBTWofish+, que faz uso principalmente das técnicas oriundas do WBAES+, permitindo alcançar uma maior resistência ao BGE-Attack, mas que no entanto, gera um criptograma distinto daquele gerado pelo Twofish.

Dentre as medidas possíveis para se averiguar a segurança desses estudos de caso, a utilizada foi a averiguação do nível de resistência que esses algoritmos apresentariam contra o BGE-Attack. Sendo o ponto de partida para essa estimativa, o custo computacional que esse ataque dispende para extrair a chave do WBAES.

Caso o WBTwofish implemente as mesmas bijeções e codificações utilizadas por Chow et al. (2003) e, devido a característica nativa da sbox do Twofish, dependente de 2 *bytes* da chave[6], pode-se obter uma segurança de 2^{40} passos contra o BGE-Attack.

Neste cálculo, são considerados os custos computacionais necessários para o BGE-Attack desfazer as codificações e bijeções mistas do WBAES, estimada em 2^{24}, com o custo necessário para que os dois *bytes* da sbox do Twofish sejam determinados pelo BGE-Attack. Assim, são necessários $2^{24} \cdot 2^{16} = 2^{40}$ passos para que a chave de cada rodada seja inferida pelo ataque.

Já o WBTwofish+, estende o WBTwofish e utiliza uma função adotada pelo WBAES+, denominada sboxgen, além também do uso de funções *one-way hash* para expansão de chave e memory-hard.

A *sboxgen*, em específico, amplia a dependência de *bytes* da *sbox* de 2 para 13 *bytes*. Isto faz com que ao invés de ter de inferir apenas 2 *bytes*, para determinar os valores de substituição da *sbox*, o que representa um custo de 2^{16} para o BGE-Attack; é necessário que o BGE-Attack obtenha os 13 *bytes* da chave manipulados pela sboxgen, ampliando a resistência individual da *sbox* do WBTwofish+ para 2^{104}; sendo 104, produto da multi-

[6]Responsáveis por tornar os valores da sbox dinâmicos

TAB. 6.1: Comparação dos algoritmos contra o BGE-Attack

Cifra	**BGE-Attack passos por round**	**Altera o criptograma?**
AES	Imediatamente	Criptograma original
WBAES	2^{24}	Não
WBAES+	2^{128}	Sim
Twofish	Imediatamente	Criptograma original
WBTwofish	2^{40}	Não
WBTwofish+	2^{128}	Sim

plicação de 13 x 8, que representa a quantidade de *bits* em cada um dos *bytes*.

A segurança alcançada pela WBTwofish+ contra o BGE-Attack, considerando-se que este faz uso das codificações e bijeções de Chow et al. (2003), é ampliada para 2^{128}. Sendo 2^{128}, obtido através dos custos computacionais de 2^{24}, necessário para que o BGE-Attack desfaça as codificações em bijeções, e do custo de 2^{104}, alcançado pela *sbox*.

Conforme mostra a tabela 6.1, que permite comparar os resultados obtidos pelos algoritmos AES, WBAES, WBAES+, Twofish, WBTwofish e WBTwofish+; a *sbox* do Twofish permite a ele e seus algoritmos derivados, alcançar uma vantagem sobre aqueles que utilizam uma *sbox* estática em um contexto *white-box*.

7 CONSIDERAÇÕES FINAIS

Tendo sido o objetivo principal desta dissertação oferecer uma análise das técnicas utilizadas para tornar o AES mais seguro em um contexto *white-box* e, demonstrar a sua aderência em um outro algoritmo, pode-se concluir que esse objetivo foi atingido; tendo em vista que, dos capítulos 2 ao 5 essa análise foi realizada e, no capítulo 6, demonstrada a sua aplicação.

Os estudos de caso apresentados, WBTwofish e WBTwofish+, apesar de não implementarem todas as técnicas mostradas ao longo da dissertação, sugerem o uso de algumas, como é o caso das bijeções misturadoras e das codificações, que permitem ao WBTwofish alcançar uma resistência teórica ao BGE-Attack de 2^{40} passos, em cada um dos *rounds*. Esse resultado confirma a afirmação de Klinec et al. (2013), ao sugerir o uso da *sbox* do Twofish no lugar da *sbox* do WBAES.

O WBTwofish+, através de alguns elementos propostos para o WBAES+, alcança uma resistência de 2^{128} passos contra o BGE-Attack, em cada um dos *rounds*, conforme demonstrado no capítulo 6.

7.1 CONTRIBUIÇÕES

Dentre as contribuições fornecidas ao longo deste trabalho, ressaltam-se, de maneira sucinta, as seguintes:

- Um levantamento da implementação *white-box*, tratando de forma mais detalhada alguns dos pontos que contribuíram para sua evolução.
- A apresentação de um conjunto de técnicas recomendáveis que permitam reforçar a estrutura do algoritmo em um contexto *white-box*, a fim de minimizar o acesso à chave criptográfica.
- Uma implementação das tabelas de pesquisa de Chow et al. (2003).
- Uma implementação do WBTwofish, que mantém o mesmo criptograma do Twofish; contudo, o torna mais resistente em um ambiente *white-box*, minimizando o vazamento de sua chave criptográfica.

- Uma implementação do WBTwofish+, que estende o funcionamento previsto para o WBTwofish e alcança em sua *sbox*, uma resistência individual de 2^{104} passos contra o BGE-Attack.

7.2 TRABALHOS FUTUROS

Dentre os diversos pontos que ficaram em aberto nesta dissertação e, que possibilitam aprimorar as implementações propostas no estudo de caso, utilizando outros elementos presentes no WBAES e no WBAES+, podem-se listar o seguintes:

- A implementação das codificações e das bijeções misturadoras para o WBTwofish e WBTwofish+, tendo em vista que ambas foram tratadas apenas conceitualmente.
- A implementação das matrizes MDS de 16x16 *bytes* para o WBTwofish+, que também só foram abordadas de forma conceitual.
- Mensurar a aplicação do BGE-Attack sobre as outras etapas do algoritmo, além da *sbox*; tanto no WBTwofish, quanto no WBTwofish+.

8 REFERÊNCIAS BIBLIOGRÁFICAS

ABRAHAO, E. Transferência de propriedade de etiquetas rfid e utilização de matrizes mds 16 x 16 na cifra de bloco aes. **Universidade Católica de Santos**, v. 1, p. 107, 2007.

BAČINSKÁ, L. **White-box attack resistant cipher based on WBAES**. 2015. 49 f. Tese (Master's degree, Information Technology Security) – Masarykova univerzita, Fakulta informatiky, Brno, 2015.

BARAK, B.; GOLDREICH, O.; IMPAGLIAZZO, R.; RUDICH, S.; SAHAI, A.; VADHAN, S. ; YANG, K. On the (im)possibility of obfuscating programs. In: KILIAN, J. (Org.). **Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 1–18. ISBN 978-3-540-44647-7.

BARRETO, P.; RIJMEN, V. **The Khazad legacy-level block cipher**. [S.l.: s.n.], 2000.

BEIMERS, M. **White-Box Cryptography**. [S.l.]: Self-published, 2014.

BILLET, O.; GILBERT, H. ; ECH-CHATBI, C. Cryptanalysis of a white box aes implementation. In: HANDSCHUH, H.; HASAN, M. A. (Org.). **Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 227–240. ISBN 978-3-540-30564-4.

CHOW, S.; EISEN, P.; JOHNSON, H. ; VAN OORSCHOT, P. C. White-box cryptography and an aes implementation. In: NYBERG, K.; HEYS, H. (Org.). **Selected Areas in Cryptography: 9th Annual International Workshop, SAC 2002 St. John's, Newfoundland, Canada, August 15–16, 2002 Revised Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 250–270. ISBN 978-3-540-36492-4.

DAEMEN, J.; KNUDSEN, L. ; RIJMEN, V. The block cipher square. In: BIHAM, E. (Org.). **Fast Software Encryption: 4th International Workshop, FSE'97 Haifa, Israel, January 20–22 1997 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 149–165. ISBN 978-3-540-69243-0.

DAEMEN, J.; RIJMEN, V. The block cipher rijndael. In: QUISQUATER, J.-J.; SCHNEIER, B. (Org.). **Smart Card Research and Applications: Third International Conference, CARDIS'98, Louvain-la-Neuve, Belgium, September 14-16, 1998. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 277–284. ISBN 978-3-540-44534-0.

FEISTEL, H.; NOTZ, W. A. ; SMITH, J. L. Some cryptographic techniques for machine-to-machine data communications. **Proceedings of the IEEE**, v. 63, n. 11, p. 1545–1554, 1975.

KERCKHOFFS, A. **La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef**. [S.l.]: Librairie militaire de L. Baudoin, 1883.

KERINS, T.; KURSAWE, K. A cautionary note on weak implementations of block ciphers. In: 1ST BENELUX WORKSHOP ON INFORMATION AND SYSTEM SECURITY (WISSEC 2006), 12., 2006. **Anais**... [S.l.: s.n.], 2006, p. 12.

KLINEC, D.; OTHERS. **White-box attack resistant cryptography**. 2013. 61 f. Tese (Master's degree, Information Technology Security) – Master's thesis, Masaryk University, Brno, Czech Republic, 2013. https://is. muni. cz/th/325219/fi_m, Brno, 2013.

KOU, W. Data encryption standards. In: _____. **Networking Security and Standards**. Boston, MA: Springer US, 1997. p. 49–67. ISBN 978-1-4615-6153-8.

LEPOINT, T.; RIVAIN, M.; DE MULDER, Y.; ROELSE, P. ; PRENEEL, B. Two attacks on a white-box aes implementation. In: SELECTED AREAS IN CRYPTOGRAPHY–SAC 2013, 20., 2013. **Anais**... [S.l.: s.n.], 2013, p. 265–285.

MUIR, J. A. A tutorial on white-box aes. In: KRANAKIS, E. (Org.). **Advances in Network Analysis and its Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 209–229. ISBN 978-3-642-30904-5.

NAKAHARA JR, J.; ABRAHAO, E. A new involutory mds matrix for the aes.. **IJ Network Security**, v. 9, n. 2, p. 109–116, 2009.

PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. **Self-published**, v. 1, p. 1–16, 2009.

RIJMEN, V.; BARRETO, P. **The Anubis block cipher**. [S.l.: s.n.], 2000.

RIJMEN, V.; DAEMEN, J.; PRENEEL, B.; BOSSELAERS, A. ; DE WIN, E. The cipher shark. In: GOLLMANN, D. (Org.). **Fast Software Encryption: Third International Workshop Cambridge, UK, February 21–23 1996 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. p. 99–111. ISBN 978-3-540-49652-6.

SCHNEIER, B.; KELSEY, J. Unbalanced feistel networks and block cipher design. In: FAST SOFTWARE ENCRYPTION, 3rd International Workshop Proceedings., 1996. **Anais**... [S.l.: s.n.], 1996, p. 121–144.

SCHNEIER, B.; KELSEY, J.; WHITING, D.; WAGNER, D.; HALL, C. ; FERGUSON, N. Twofish: A 128-bit block cipher. **NIST AES Proposal**, v. 15, p. 68, 1998.

SHANNON, C. E. Communication theory of secrecy systems. **Bell system technical journal**, v. 28, n. 4, p. 656–715, 1949.

SINGH, S. **The code book: the science of secrecy from ancient Egypt to quantum cryptography**. [S.l.]: Anchor, 2011.

SINGLETON, R. Maximum distance q-nary codes. **IEEE Transactions on Information Theory**, v. 10, n. 2, p. 116–118, 1964.

STANDARD, N.-F. Announcing the advanced encryption standard (aes). **Federal Information Processing Standards Publication**, v. 197, p. 1–51, 2001.

XIAO, J.; ZHOU, Y. Generating large non-singular matrices over an arbitrary field with blocks of full rank. **IACR Eprint archive**, v. 1, p. 6, 2002.

9 APÊNDICES

```
// WBTwofish.java
public class WBTwofish {

    public static byte[] ToByteArray(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i + 1), 16));
        }
        return data;
    }

    private static String hex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bytes.length; i++)
            sb.append(String.format("%02X ", bytes[i]));
        return sb.toString();
    }

    public static void main(String[] args) {
        Twofish_Algorithm twofishAlgorithm = new Twofish_Algorithm();
        // First parameter: key | Second parameter: plaintext
        byte[] cipherText =
            twofishAlgorithm.encrypt(ToByteArray("54776f46697368206973206e69636520"),
            ToByteArray("57656c636f6d52054776f46669736820"));
        byte[] plainText =
            twofishAlgorithm.decrypt(ToByteArray("54776f46697368206973206e69636520"),
            cipherText);
    }

}
```

```
// UtilWBTwofish.java
import java.security.MessageDigest;

public class UtilWBTwofish {

    protected static final byte q8x8[][] = new byte[][]{
        {
            (byte) 0xA9, (byte) 0x67, (byte) 0xB3, (byte) 0xE8, (byte) 0x04,
                (byte) 0xFD, (byte) 0xA3, (byte) 0x76,
            (byte) 0x9A, (byte) 0x92, (byte) 0x80, (byte) 0x78, (byte) 0xE4,
                (byte) 0xDD, (byte) 0xD1, (byte) 0x38,
            (byte) 0x0D, (byte) 0xC6, (byte) 0x35, (byte) 0x98, (byte) 0x18,
                (byte) 0xF7, (byte) 0xEC, (byte) 0x6C,
            (byte) 0x43, (byte) 0x75, (byte) 0x37, (byte) 0x26, (byte) 0xFA,
                (byte) 0x13, (byte) 0x94, (byte) 0x48,
            (byte) 0xF2, (byte) 0xD0, (byte) 0x8B, (byte) 0x30, (byte) 0x84,
                (byte) 0x54, (byte) 0xDF, (byte) 0x23,
            (byte) 0x19, (byte) 0x5B, (byte) 0x3D, (byte) 0x59, (byte) 0xF3,
                (byte) 0xAE, (byte) 0xA2, (byte) 0x82,
            (byte) 0x63, (byte) 0x01, (byte) 0x83, (byte) 0x2E, (byte) 0xD9,
                (byte) 0x51, (byte) 0x9B, (byte) 0x7C,
            (byte) 0xA6, (byte) 0xEB, (byte) 0xA5, (byte) 0xBE, (byte) 0x16,
                (byte) 0x0C, (byte) 0xE3, (byte) 0x61,
            (byte) 0xC0, (byte) 0x8C, (byte) 0x3A, (byte) 0xF5, (byte) 0x73,
                (byte) 0x2C, (byte) 0x25, (byte) 0x0B,
            (byte) 0xBB, (byte) 0x4E, (byte) 0x89, (byte) 0x6B, (byte) 0x53,
                (byte) 0x6A, (byte) 0xB4, (byte) 0xF1,
            (byte) 0xE1, (byte) 0xE6, (byte) 0xBD, (byte) 0x45, (byte) 0xE2,
                (byte) 0xF4, (byte) 0xB6, (byte) 0x66,
            (byte) 0xCC, (byte) 0x95, (byte) 0x03, (byte) 0x56, (byte) 0xD4,
                (byte) 0x1C, (byte) 0x1E, (byte) 0xD7,
            (byte) 0xFB, (byte) 0xC3, (byte) 0x8E, (byte) 0xB5, (byte) 0xE9,
                (byte) 0xCF, (byte) 0xBF, (byte) 0xBA,
            (byte) 0xEA, (byte) 0x77, (byte) 0x39, (byte) 0xAF, (byte) 0x33,
                (byte) 0xC9, (byte) 0x62, (byte) 0x71,
            (byte) 0x81, (byte) 0x79, (byte) 0x09, (byte) 0xAD, (byte) 0x24,
```

```
        (byte) 0xCD, (byte) 0xF9, (byte) 0xD8,
    (byte) 0xE5, (byte) 0xC5, (byte) 0xB9, (byte) 0x4D, (byte) 0x44,
        (byte) 0x08, (byte) 0x86, (byte) 0xE7,
    (byte) 0xA1, (byte) 0x1D, (byte) 0xAA, (byte) 0xED, (byte) 0x06,
        (byte) 0x70, (byte) 0xB2, (byte) 0xD2,
    (byte) 0x41, (byte) 0x7B, (byte) 0xA0, (byte) 0x11, (byte) 0x31,
        (byte) 0xC2, (byte) 0x27, (byte) 0x90,
    (byte) 0x20, (byte) 0xF6, (byte) 0x60, (byte) 0xFF, (byte) 0x96,
        (byte) 0x5C, (byte) 0xB1, (byte) 0xAB,
    (byte) 0x9E, (byte) 0x9C, (byte) 0x52, (byte) 0x1B, (byte) 0x5F,
        (byte) 0x93, (byte) 0x0A, (byte) 0xEF,
    (byte) 0x91, (byte) 0x85, (byte) 0x49, (byte) 0xEE, (byte) 0x2D,
        (byte) 0x4F, (byte) 0x8F, (byte) 0x3B,
    (byte) 0x47, (byte) 0x87, (byte) 0x6D, (byte) 0x46, (byte) 0xD6,
        (byte) 0x3E, (byte) 0x69, (byte) 0x64,
    (byte) 0x2A, (byte) 0xCE, (byte) 0xCB, (byte) 0x2F, (byte) 0xFC,
        (byte) 0x97, (byte) 0x05, (byte) 0x7A,
    (byte) 0xAC, (byte) 0x7F, (byte) 0xD5, (byte) 0x1A, (byte) 0x4B,
        (byte) 0x0E, (byte) 0xA7, (byte) 0x5A,
    (byte) 0x28, (byte) 0x14, (byte) 0x3F, (byte) 0x29, (byte) 0x88,
        (byte) 0x3C, (byte) 0x4C, (byte) 0x02,
    (byte) 0xB8, (byte) 0xDA, (byte) 0xB0, (byte) 0x17, (byte) 0x55,
        (byte) 0x1F, (byte) 0x8A, (byte) 0x7D,
    (byte) 0x57, (byte) 0xC7, (byte) 0x8D, (byte) 0x74, (byte) 0xB7,
        (byte) 0xC4, (byte) 0x9F, (byte) 0x72,
    (byte) 0x7E, (byte) 0x15, (byte) 0x22, (byte) 0x12, (byte) 0x58,
        (byte) 0x07, (byte) 0x99, (byte) 0x34,
    (byte) 0x6E, (byte) 0x50, (byte) 0xDE, (byte) 0x68, (byte) 0x65,
        (byte) 0xBC, (byte) 0xDB, (byte) 0xF8,
    (byte) 0xC8, (byte) 0xA8, (byte) 0x2B, (byte) 0x40, (byte) 0xDC,
        (byte) 0xFE, (byte) 0x32, (byte) 0xA4,
    (byte) 0xCA, (byte) 0x10, (byte) 0x21, (byte) 0xF0, (byte) 0xD3,
        (byte) 0x5D, (byte) 0x0F, (byte) 0x00,
    (byte) 0x6F, (byte) 0x9D, (byte) 0x36, (byte) 0x42, (byte) 0x4A,
        (byte) 0x5E, (byte) 0xC1, (byte) 0xE0
},
{
```

```
(byte) 0x75, (byte) 0xF3, (byte) 0xC6, (byte) 0xF4, (byte) 0xDB,
    (byte) 0x7B, (byte) 0xFB, (byte) 0xC8,
(byte) 0x4A, (byte) 0xD3, (byte) 0xE6, (byte) 0x6B, (byte) 0x45,
    (byte) 0x7D, (byte) 0xE8, (byte) 0x4B,
(byte) 0xD6, (byte) 0x32, (byte) 0xD8, (byte) 0xFD, (byte) 0x37,
    (byte) 0x71, (byte) 0xF1, (byte) 0xE1,
(byte) 0x30, (byte) 0x0F, (byte) 0xF8, (byte) 0x1B, (byte) 0x87,
    (byte) 0xFA, (byte) 0x06, (byte) 0x3F,
(byte) 0x5E, (byte) 0xBA, (byte) 0xAE, (byte) 0x5B, (byte) 0x8A,
    (byte) 0x00, (byte) 0xBC, (byte) 0x9D,
(byte) 0x6D, (byte) 0xC1, (byte) 0xB1, (byte) 0x0E, (byte) 0x80,
    (byte) 0x5D, (byte) 0xD2, (byte) 0xD5,
(byte) 0xA0, (byte) 0x84, (byte) 0x07, (byte) 0x14, (byte) 0xB5,
    (byte) 0x90, (byte) 0x2C, (byte) 0xA3,
(byte) 0xB2, (byte) 0x73, (byte) 0x4C, (byte) 0x54, (byte) 0x92,
    (byte) 0x74, (byte) 0x36, (byte) 0x51,
(byte) 0x38, (byte) 0xB0, (byte) 0xBD, (byte) 0x5A, (byte) 0xFC,
    (byte) 0x60, (byte) 0x62, (byte) 0x96,
(byte) 0x6C, (byte) 0x42, (byte) 0xF7, (byte) 0x10, (byte) 0x7C,
    (byte) 0x28, (byte) 0x27, (byte) 0x8C,
(byte) 0x13, (byte) 0x95, (byte) 0x9C, (byte) 0xC7, (byte) 0x24,
    (byte) 0x46, (byte) 0x3B, (byte) 0x70,
(byte) 0xCA, (byte) 0xE3, (byte) 0x85, (byte) 0xCB, (byte) 0x11,
    (byte) 0xD0, (byte) 0x93, (byte) 0xB8,
(byte) 0xA6, (byte) 0x83, (byte) 0x20, (byte) 0xFF, (byte) 0x9F,
    (byte) 0x77, (byte) 0xC3, (byte) 0xCC,
(byte) 0x03, (byte) 0x6F, (byte) 0x08, (byte) 0xBF, (byte) 0x40,
    (byte) 0xE7, (byte) 0x2B, (byte) 0xE2,
(byte) 0x79, (byte) 0x0C, (byte) 0xAA, (byte) 0x82, (byte) 0x41,
    (byte) 0x3A, (byte) 0xEA, (byte) 0xB9,
(byte) 0xE4, (byte) 0x9A, (byte) 0xA4, (byte) 0x97, (byte) 0x7E,
    (byte) 0xDA, (byte) 0x7A, (byte) 0x17,
(byte) 0x66, (byte) 0x94, (byte) 0xA1, (byte) 0x1D, (byte) 0x3D,
    (byte) 0xF0, (byte) 0xDE, (byte) 0xB3,
(byte) 0x0B, (byte) 0x72, (byte) 0xA7, (byte) 0x1C, (byte) 0xEF,
    (byte) 0xD1, (byte) 0x53, (byte) 0x3E,
(byte) 0x8F, (byte) 0x33, (byte) 0x26, (byte) 0x5F, (byte) 0xEC,
```

```
            (byte) 0x76, (byte) 0x2A, (byte) 0x49,
        (byte) 0x81, (byte) 0x88, (byte) 0xEE, (byte) 0x21, (byte) 0xC4,
            (byte) 0x1A, (byte) 0xEB, (byte) 0xD9,
        (byte) 0xC5, (byte) 0x39, (byte) 0x99, (byte) 0xCD, (byte) 0xAD,
            (byte) 0x31, (byte) 0x8B, (byte) 0x01,
        (byte) 0x18, (byte) 0x23, (byte) 0xDD, (byte) 0x1F, (byte) 0x4E,
            (byte) 0x2D, (byte) 0xF9, (byte) 0x48,
        (byte) 0x4F, (byte) 0xF2, (byte) 0x65, (byte) 0x8E, (byte) 0x78,
            (byte) 0x5C, (byte) 0x58, (byte) 0x19,
        (byte) 0x8D, (byte) 0xE5, (byte) 0x98, (byte) 0x57, (byte) 0x67,
            (byte) 0x7F, (byte) 0x05, (byte) 0x64,
        (byte) 0xAF, (byte) 0x63, (byte) 0xB6, (byte) 0xFE, (byte) 0xF5,
            (byte) 0xB7, (byte) 0x3C, (byte) 0xA5,
        (byte) 0xCE, (byte) 0xE9, (byte) 0x68, (byte) 0x44, (byte) 0xE0,
            (byte) 0x4D, (byte) 0x43, (byte) 0x69,
        (byte) 0x29, (byte) 0x2E, (byte) 0xAC, (byte) 0x15, (byte) 0x59,
            (byte) 0xA8, (byte) 0x0A, (byte) 0x9E,
        (byte) 0x6E, (byte) 0x47, (byte) 0xDF, (byte) 0x34, (byte) 0x35,
            (byte) 0x6A, (byte) 0xCF, (byte) 0xDC,
        (byte) 0x22, (byte) 0xC9, (byte) 0xC0, (byte) 0x9B, (byte) 0x89,
            (byte) 0xD4, (byte) 0xED, (byte) 0xAB,
        (byte) 0x12, (byte) 0xA2, (byte) 0x0D, (byte) 0x52, (byte) 0xBB,
            (byte) 0x02, (byte) 0x2F, (byte) 0xA9,
        (byte) 0xD7, (byte) 0x61, (byte) 0x1E, (byte) 0xB4, (byte) 0x50,
            (byte) 0x04, (byte) 0xF6, (byte) 0xC2,
        (byte) 0x16, (byte) 0x25, (byte) 0x86, (byte) 0x56, (byte) 0x55,
            (byte) 0x09, (byte) 0xBE, (byte) 0x91
    }
};

public static byte ConcatBytes(byte[] array, int startPoint, int nest) {
    byte outputByte = 0;
    for (int i = 0; i < nest - 1; i++) {
        outputByte = (byte) ((array[startPoint] << 8) + array[startPoint +
            1]);
    }
    return outputByte;
```

```
}

public static byte[] GetByteArrayFromInt(int x, int bitsNumber) {
    byte[] result = new byte[bitsNumber / 8];
    for (int i = 0, j = 0; i < bitsNumber / 8; i++, j += 8) {
        result[i] = (byte) (x >>> j);
    }
    return result;
}

public static byte[] ConcatByteArrayFromInt(int[] inputs, int bitsNumber,
    int outputSize) {
    byte[] result = new byte[outputSize];
    int bitsToCopy = bitsNumber / 8;
    for (int i = 0; i < inputs.length; i++) {
        System.arraycopy(GetByteArrayFromInt(inputs[i], bitsNumber), 0,
            result, bitsToCopy * i, bitsToCopy);
    }
    return result;
}

public static int[] ConverFromByteArrayToIntArray(byte[] inputArray) {
    int[] outputArray = new int[inputArray.length];

    for (int i = 0; i < inputArray.length; i++) {
        outputArray[i] = inputArray[i] & 0xff;
    }
    return outputArray;
}

public static byte[] keyBytesDerivation(byte[] key) {

    byte s[] = new byte[64];

    try {
        MessageDigest md = MessageDigest.getInstance("SHA-512");
        md.update(key);
```

```
        s = md.digest();
    } catch (Exception e) { //NoSuchAlgorithmException
        System.out.println("Problem with SHA512 in keyBytesDerivation (used
            in createKeyDependentSboxes).");
    }

    return s;
}

private static byte[] hashFunction(byte[] input, byte[] salt, int size, int
    sc_N, int sc_r, int sc_p, int n_sha) {
  int i;
  byte[] tmpInput = new byte[input.length];
  System.arraycopy(input, 0, tmpInput, 0, input.length);

  for(i = 0; i < n_sha; i++) {
     try {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        md.update(tmpInput);
        tmpInput = md.digest();
     } catch(Exception e) { //NoSuchAlgorithmException
        System.out.println("Problem with SHA256 in hashFunction (used in
            hashChain).");
     }
  }

  return SCrypt.generate(tmpInput, salt, sc_N, sc_r, sc_p, size);
}

public static byte[] hashChain(byte[] key, int size, String saltString,
    boolean debug) {

    int currentSize = 0;
    int i;
    int roundKeysSize = 176;
    int roundsNum = 8;
```

```
    byte[] roundKeys = new byte[roundKeysSize];
    byte[] tmpKey = new byte[size];
    byte[] salt = saltString.getBytes();
    int scrypt_N = 16384;
    int scrypt_R = 8;
    int scrypt_P = 1;

    System.arraycopy(key, 0, tmpKey, 0, size);

    for (i = 0; i < roundsNum + 1; i++) {

        if (i == 0) {
            tmpKey = hashFunction(key, salt, size, scrypt_N, scrypt_R,
                scrypt_P, 16);
        } else {
            byte[] hashInput = new byte[2 * size];
            System.arraycopy(tmpKey, 0, hashInput, 0, size);
            System.arraycopy(key, 0, hashInput, size, size);

            tmpKey = hashFunction(hashInput, salt, size, scrypt_N, scrypt_R,
                scrypt_P, 16);
        }

        System.arraycopy(tmpKey, 0, roundKeys, currentSize, size);
        currentSize += size;
    }
    return roundKeys;
}

public static byte[] createRoundKeysForSboxes(byte[] key, int size) {
    byte[] sboxSalt = "WBTwofish+SB".getBytes();
    byte[] constantSalt = "WBTWofishPlusalt".getBytes();
    byte[] input = new byte[key.length + constantSalt.length];
    System.arraycopy(key, 0, input, 0, key.length);
    System.arraycopy(constantSalt, 0, input, key.length,
        constantSalt.length);
    byte[] roundKeysForSboxes = hashChain(input, size, "WBTWofishPlusalt",
```

```
            false);
        return roundKeysForSboxes;
    }

    public static int sboxgen(int j, int l, int x, byte[] key_bytes) {

        int qs[][] = new int[][]{{0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1},
        {1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0},
        {0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1},
        {1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1}};

        if (l == 0) {
            return (int) q8x8[qs[j][0]][x] & 0xff;
        } else {
            return (int) q8x8[qs[j][l]][sboxgen(j, l - 1, x, key_bytes) ^ ((int)
                key_bytes[4 * (l - 1) + j] & 0xff)] & 0xff;
        }
    }

}
```

```
//Twofish_Properties.java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.PrintStream;
import java.util.Enumeration;
import java.util.Properties;


/**
 * <b>Copyright</b> &copy; 1997, 1998
 * <a href="http://www.systemics.com/">Systemics Ltd</a> on behalf of the
 * <a href="http://www.systemics.com/docs/cryptix/">Cryptix Development
 * Team</a>.
```

```
 * <br>All rights reserved.<p>
 *
 * <b>$Revision: $</b>
 *
 * @author David Hopwood
 * @author Jill Baker
 * @author Raif S. Naffah
 */

public class Twofish_Properties // implicit no-argument constructor
{
    static final boolean GLOBAL_DEBUG = false;

    static final String ALGORITHM = "Twofish";
    static final double VERSION = 0.2;
    static final String FULL_NAME = ALGORITHM + " ver. " + VERSION;
    static final String NAME = "Twofish_Properties";

    static final Properties properties = new Properties();

    /**
     * Default properties in case .properties file was not found.
     */
    private static final String[][] DEFAULT_PROPERTIES = {
        {"Trace.Twofish_Algorithm", "true"},
        {"Debug.Level.*", "1"},
        {"Debug.Level.Twofish_Algorithm", "9"},};

    static {
        if (GLOBAL_DEBUG) {
            System.err.println(">>> " + NAME + ": Looking for " + ALGORITHM + "
                properties");
        }
        String it = ALGORITHM + ".properties";
        InputStream is = Twofish_Properties.class.getResourceAsStream(it);
        boolean ok = is != null;
        if (ok) {
```

```
            try {
                properties.load(is);
                is.close();
                if (GLOBAL_DEBUG) {
                    System.err.println(">>> " + NAME + ": Properties file loaded
                        OK...");
                }
            } catch (Exception x) {
                ok = false;
            }
        }
        if (!ok) {
            if (GLOBAL_DEBUG) {
                System.err.println(">>> " + NAME + ": WARNING: Unable to load \""
                    + it + "\" from CLASSPATH.");
            }
            if (GLOBAL_DEBUG) {
                System.err.println(">>> " + NAME + ":     Will use default
                    values instead...");
            }
            int n = DEFAULT_PROPERTIES.length;
            for (int i = 0; i < n; i++) {
                properties.put(
                        DEFAULT_PROPERTIES[i][0], DEFAULT_PROPERTIES[i][1]);
            }
            if (GLOBAL_DEBUG) {
                System.err.println(">>> " + NAME + ": Default properties now
                    set...");
            }
        }
    }

    public static String getProperty(String key) {
        return properties.getProperty(key);
    }

    public static String getProperty(String key, String value) {
```

```
    return properties.getProperty(key, value);
}

public static void list(PrintStream out) {
    list(new PrintWriter(out, true));
}

public static void list(PrintWriter out) {
    out.println("#");
    out.println("# ----- Begin " + ALGORITHM + " properties -----");
    out.println("#");
    String key, value;
    Enumeration enumE = properties.propertyNames();
    while(enumE.hasMoreElements()) {
        key = (String)
        enumE.nextElement();
        value = getProperty(key);
        out.println(key + " = " + value);
    }
    out.println("#");
    out.println("# ----- End " + ALGORITHM + " properties -----");
}

public static Enumeration propertyNames() {
    return properties.propertyNames();
}

static boolean isTraceable(String label) {
    String s = getProperty("Trace." + label);
    if (s == null) {
        return false;
    }
    return new Boolean(s).booleanValue();
}

static int getLevel(String label) {
    String s = getProperty("Debug.Level." + label);
```

```
        if (s == null) {
            s = getProperty("Debug.Level.*");
            if (s == null) {
                return 0;
            }
        }
        try {
            return Integer.parseInt(s);
        } catch (NumberFormatException e) {
            return 0;
        }
    }

    static PrintWriter getOutput() {
        PrintWriter pw;
        String name = getProperty("Output");
        if (name != null && name.equals("out")) {
            pw = new PrintWriter(System.out, true);
        } else {
            pw = new PrintWriter(System.err, true);
        }
        return pw;
    }
}
```

```
//WBTwofish.java

import java.io.PrintWriter;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.security.InvalidKeyException;
import java.security.SecureRandom;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.bouncycastle.crypto.generators.SCrypt;
```

```
//..............................................................................
 * Twofish was submitted by Bruce Schneier, Doug Whiting, John Kelsey, Chris
 * Hall and David Wagner.<p>
 *
 * Reference:<ol>
 * <li>TWOFISH2.C -- Optimized C API calls for TWOFISH AES submission, Version
 * 1.00, April 1998, by Doug Whiting.</ol><p>
 *
 * <b>Copyright</b> &copy; 1998
 * <a href="http://www.systemics.com/">Systemics Ltd</a> on behalf of the
 * <a href="http://www.systemics.com/docs/cryptix/">Cryptix Development
 * Team</a>.
 * <br>All rights reserved.<p>
 *
 * <b>$Revision: $</b>
 *
 * @author Raif S. Naffah
 */

public final class Twofish_Algorithm // implicit no-argument constructor
{

   static final String NAME = "Twofish_Algorithm";
   static final boolean IN = true, OUT = false;

   static final boolean DEBUG = Twofish_Properties.GLOBAL_DEBUG;
   static final int debuglevel = DEBUG ? Twofish_Properties.getLevel(NAME) : 0;
   static final PrintWriter err = DEBUG ? Twofish_Properties.getOutput() :
       null;

   static final boolean TRACE = Twofish_Properties.isTraceable(NAME);

   static void debug(String s) {
       err.println(">>> " + NAME + ": " + s);
   }

   static void trace(boolean in, String s) {
```

```
    if (TRACE) {
        err.println((in ? "==> " : "<== ") + NAME + "." + s);
    }
}

static void trace(String s) {
    if (TRACE) {
        err.println("<=> " + NAME + "." + s);
    }
}

static final int BLOCK_SIZE = 16; // bytes in a data-block
private static final int ROUNDS = 16;
private static final int MAX_ROUNDS = 16; // max # rounds (for allocating
    subkeys)

private static final int INPUT_WHITEN = 0;
private static final int OUTPUT_WHITEN = INPUT_WHITEN + BLOCK_SIZE / 4;
private static final int ROUND_SUBKEYS = OUTPUT_WHITEN + BLOCK_SIZE / 4; //
    2*(# rounds)

private static final int TOTAL_SUBKEYS = ROUND_SUBKEYS + 2 * MAX_ROUNDS;

private static final int SK_STEP = 0x02020202;
private static final int SK_BUMP = 0x01010101;
private static final int SK_ROTL = 9;

/**
 * Fixed 8x8 permutation S-boxes
 */
private static final byte[][] P = new byte[][]{
    { // p0
        (byte) 0xA9, (byte) 0x67, (byte) 0xB3, (byte) 0xE8,
        (byte) 0x04, (byte) 0xFD, (byte) 0xA3, (byte) 0x76,
        (byte) 0x9A, (byte) 0x92, (byte) 0x80, (byte) 0x78,
        (byte) 0xE4, (byte) 0xDD, (byte) 0xD1, (byte) 0x38,
        (byte) 0x0D, (byte) 0xC6, (byte) 0x35, (byte) 0x98,
```

```
(byte) 0x18, (byte) 0xF7, (byte) 0xEC, (byte) 0x6C,
(byte) 0x43, (byte) 0x75, (byte) 0x37, (byte) 0x26,
(byte) 0xFA, (byte) 0x13, (byte) 0x94, (byte) 0x48,
(byte) 0xF2, (byte) 0xD0, (byte) 0x8B, (byte) 0x30,
(byte) 0x84, (byte) 0x54, (byte) 0xDF, (byte) 0x23,
(byte) 0x19, (byte) 0x5B, (byte) 0x3D, (byte) 0x59,
(byte) 0xF3, (byte) 0xAE, (byte) 0xA2, (byte) 0x82,
(byte) 0x63, (byte) 0x01, (byte) 0x83, (byte) 0x2E,
(byte) 0xD9, (byte) 0x51, (byte) 0x9B, (byte) 0x7C,
(byte) 0xA6, (byte) 0xEB, (byte) 0xA5, (byte) 0xBE,
(byte) 0x16, (byte) 0x0C, (byte) 0xE3, (byte) 0x61,
(byte) 0xC0, (byte) 0x8C, (byte) 0x3A, (byte) 0xF5,
(byte) 0x73, (byte) 0x2C, (byte) 0x25, (byte) 0x0B,
(byte) 0xBB, (byte) 0x4E, (byte) 0x89, (byte) 0x6B,
(byte) 0x53, (byte) 0x6A, (byte) 0xB4, (byte) 0xF1,
(byte) 0xE1, (byte) 0xE6, (byte) 0xBD, (byte) 0x45,
(byte) 0xE2, (byte) 0xF4, (byte) 0xB6, (byte) 0x66,
(byte) 0xCC, (byte) 0x95, (byte) 0x03, (byte) 0x56,
(byte) 0xD4, (byte) 0x1C, (byte) 0x1E, (byte) 0xD7,
(byte) 0xFB, (byte) 0xC3, (byte) 0x8E, (byte) 0xB5,
(byte) 0xE9, (byte) 0xCF, (byte) 0xBF, (byte) 0xBA,
(byte) 0xEA, (byte) 0x77, (byte) 0x39, (byte) 0xAF,
(byte) 0x33, (byte) 0xC9, (byte) 0x62, (byte) 0x71,
(byte) 0x81, (byte) 0x79, (byte) 0x09, (byte) 0xAD,
(byte) 0x24, (byte) 0xCD, (byte) 0xF9, (byte) 0xD8,
(byte) 0xE5, (byte) 0xC5, (byte) 0xB9, (byte) 0x4D,
(byte) 0x44, (byte) 0x08, (byte) 0x86, (byte) 0xE7,
(byte) 0xA1, (byte) 0x1D, (byte) 0xAA, (byte) 0xED,
(byte) 0x06, (byte) 0x70, (byte) 0xB2, (byte) 0xD2,
(byte) 0x41, (byte) 0x7B, (byte) 0xA0, (byte) 0x11,
(byte) 0x31, (byte) 0xC2, (byte) 0x27, (byte) 0x90,
(byte) 0x20, (byte) 0xF6, (byte) 0x60, (byte) 0xFF,
(byte) 0x96, (byte) 0x5C, (byte) 0xB1, (byte) 0xAB,
(byte) 0x9E, (byte) 0x9C, (byte) 0x52, (byte) 0x1B,
(byte) 0x5F, (byte) 0x93, (byte) 0x0A, (byte) 0xEF,
(byte) 0x91, (byte) 0x85, (byte) 0x49, (byte) 0xEE,
(byte) 0x2D, (byte) 0x4F, (byte) 0x8F, (byte) 0x3B,
```

```
    (byte) 0x47, (byte) 0x87, (byte) 0x6D, (byte) 0x46,
    (byte) 0xD6, (byte) 0x3E, (byte) 0x69, (byte) 0x64,
    (byte) 0x2A, (byte) 0xCE, (byte) 0xCB, (byte) 0x2F,
    (byte) 0xFC, (byte) 0x97, (byte) 0x05, (byte) 0x7A,
    (byte) 0xAC, (byte) 0x7F, (byte) 0xD5, (byte) 0x1A,
    (byte) 0x4B, (byte) 0x0E, (byte) 0xA7, (byte) 0x5A,
    (byte) 0x28, (byte) 0x14, (byte) 0x3F, (byte) 0x29,
    (byte) 0x88, (byte) 0x3C, (byte) 0x4C, (byte) 0x02,
    (byte) 0xB8, (byte) 0xDA, (byte) 0xB0, (byte) 0x17,
    (byte) 0x55, (byte) 0x1F, (byte) 0x8A, (byte) 0x7D,
    (byte) 0x57, (byte) 0xC7, (byte) 0x8D, (byte) 0x74,
    (byte) 0xB7, (byte) 0xC4, (byte) 0x9F, (byte) 0x72,
    (byte) 0x7E, (byte) 0x15, (byte) 0x22, (byte) 0x12,
    (byte) 0x58, (byte) 0x07, (byte) 0x99, (byte) 0x34,
    (byte) 0x6E, (byte) 0x50, (byte) 0xDE, (byte) 0x68,
    (byte) 0x65, (byte) 0xBC, (byte) 0xDB, (byte) 0xF8,
    (byte) 0xC8, (byte) 0xA8, (byte) 0x2B, (byte) 0x40,
    (byte) 0xDC, (byte) 0xFE, (byte) 0x32, (byte) 0xA4,
    (byte) 0xCA, (byte) 0x10, (byte) 0x21, (byte) 0xF0,
    (byte) 0xD3, (byte) 0x5D, (byte) 0x0F, (byte) 0x00,
    (byte) 0x6F, (byte) 0x9D, (byte) 0x36, (byte) 0x42,
    (byte) 0x4A, (byte) 0x5E, (byte) 0xC1, (byte) 0xE0
},
{ // p1
    (byte) 0x75, (byte) 0xF3, (byte) 0xC6, (byte) 0xF4,
    (byte) 0xDB, (byte) 0x7B, (byte) 0xFB, (byte) 0xC8,
    (byte) 0x4A, (byte) 0xD3, (byte) 0xE6, (byte) 0x6B,
    (byte) 0x45, (byte) 0x7D, (byte) 0xE8, (byte) 0x4B,
    (byte) 0xD6, (byte) 0x32, (byte) 0xD8, (byte) 0xFD,
    (byte) 0x37, (byte) 0x71, (byte) 0xF1, (byte) 0xE1,
    (byte) 0x30, (byte) 0x0F, (byte) 0xF8, (byte) 0x1B,
    (byte) 0x87, (byte) 0xFA, (byte) 0x06, (byte) 0x3F,
    (byte) 0x5E, (byte) 0xBA, (byte) 0xAE, (byte) 0x5B,
    (byte) 0x8A, (byte) 0x00, (byte) 0xBC, (byte) 0x9D,
    (byte) 0x6D, (byte) 0xC1, (byte) 0xB1, (byte) 0x0E,
    (byte) 0x80, (byte) 0x5D, (byte) 0xD2, (byte) 0xD5,
    (byte) 0xA0, (byte) 0x84, (byte) 0x07, (byte) 0x14,
```

```
(byte) 0xB5, (byte) 0x90, (byte) 0x2C, (byte) 0xA3,
(byte) 0xB2, (byte) 0x73, (byte) 0x4C, (byte) 0x54,
(byte) 0x92, (byte) 0x74, (byte) 0x36, (byte) 0x51,
(byte) 0x38, (byte) 0xB0, (byte) 0xBD, (byte) 0x5A,
(byte) 0xFC, (byte) 0x60, (byte) 0x62, (byte) 0x96,
(byte) 0x6C, (byte) 0x42, (byte) 0xF7, (byte) 0x10,
(byte) 0x7C, (byte) 0x28, (byte) 0x27, (byte) 0x8C,
(byte) 0x13, (byte) 0x95, (byte) 0x9C, (byte) 0xC7,
(byte) 0x24, (byte) 0x46, (byte) 0x3B, (byte) 0x70,
(byte) 0xCA, (byte) 0xE3, (byte) 0x85, (byte) 0xCB,
(byte) 0x11, (byte) 0xD0, (byte) 0x93, (byte) 0xB8,
(byte) 0xA6, (byte) 0x83, (byte) 0x20, (byte) 0xFF,
(byte) 0x9F, (byte) 0x77, (byte) 0xC3, (byte) 0xCC,
(byte) 0x03, (byte) 0x6F, (byte) 0x08, (byte) 0xBF,
(byte) 0x40, (byte) 0xE7, (byte) 0x2B, (byte) 0xE2,
(byte) 0x79, (byte) 0x0C, (byte) 0xAA, (byte) 0x82,
(byte) 0x41, (byte) 0x3A, (byte) 0xEA, (byte) 0xB9,
(byte) 0xE4, (byte) 0x9A, (byte) 0xA4, (byte) 0x97,
(byte) 0x7E, (byte) 0xDA, (byte) 0x7A, (byte) 0x17,
(byte) 0x66, (byte) 0x94, (byte) 0xA1, (byte) 0x1D,
(byte) 0x3D, (byte) 0xF0, (byte) 0xDE, (byte) 0xB3,
(byte) 0x0B, (byte) 0x72, (byte) 0xA7, (byte) 0x1C,
(byte) 0xEF, (byte) 0xD1, (byte) 0x53, (byte) 0x3E,
(byte) 0x8F, (byte) 0x33, (byte) 0x26, (byte) 0x5F,
(byte) 0xEC, (byte) 0x76, (byte) 0x2A, (byte) 0x49,
(byte) 0x81, (byte) 0x88, (byte) 0xEE, (byte) 0x21,
(byte) 0xC4, (byte) 0x1A, (byte) 0xEB, (byte) 0xD9,
(byte) 0xC5, (byte) 0x39, (byte) 0x99, (byte) 0xCD,
(byte) 0xAD, (byte) 0x31, (byte) 0x8B, (byte) 0x01,
(byte) 0x18, (byte) 0x23, (byte) 0xDD, (byte) 0x1F,
(byte) 0x4E, (byte) 0x2D, (byte) 0xF9, (byte) 0x48,
(byte) 0x4F, (byte) 0xF2, (byte) 0x65, (byte) 0x8E,
(byte) 0x78, (byte) 0x5C, (byte) 0x58, (byte) 0x19,
(byte) 0x8D, (byte) 0xE5, (byte) 0x98, (byte) 0x57,
(byte) 0x67, (byte) 0x7F, (byte) 0x05, (byte) 0x64,
(byte) 0xAF, (byte) 0x63, (byte) 0xB6, (byte) 0xFE,
(byte) 0xF5, (byte) 0xB7, (byte) 0x3C, (byte) 0xA5,
```

```
        (byte) 0xCE, (byte) 0xE9, (byte) 0x68, (byte) 0x44,
        (byte) 0xE0, (byte) 0x4D, (byte) 0x43, (byte) 0x69,
        (byte) 0x29, (byte) 0x2E, (byte) 0xAC, (byte) 0x15,
        (byte) 0x59, (byte) 0xA8, (byte) 0x0A, (byte) 0x9E,
        (byte) 0x6E, (byte) 0x47, (byte) 0xDF, (byte) 0x34,
        (byte) 0x35, (byte) 0x6A, (byte) 0xCF, (byte) 0xDC,
        (byte) 0x22, (byte) 0xC9, (byte) 0xC0, (byte) 0x9B,
        (byte) 0x89, (byte) 0xD4, (byte) 0xED, (byte) 0xAB,
        (byte) 0x12, (byte) 0xA2, (byte) 0x0D, (byte) 0x52,
        (byte) 0xBB, (byte) 0x02, (byte) 0x2F, (byte) 0xA9,
        (byte) 0xD7, (byte) 0x61, (byte) 0x1E, (byte) 0xB4,
        (byte) 0x50, (byte) 0x04, (byte) 0xF6, (byte) 0xC2,
        (byte) 0x16, (byte) 0x25, (byte) 0x86, (byte) 0x56,
        (byte) 0x55, (byte) 0x09, (byte) 0xBE, (byte) 0x91
    }
};

private static final int P_00 = 1;
private static final int P_01 = 0;
private static final int P_02 = 0;
private static final int P_03 = P_01 ^ 1;
private static final int P_04 = 1;

private static final int P_10 = 0;
private static final int P_11 = 0;
private static final int P_12 = 1;
private static final int P_13 = P_11 ^ 1;
private static final int P_14 = 0;

private static final int P_20 = 1;
private static final int P_21 = 1;
private static final int P_22 = 0;
private static final int P_23 = P_21 ^ 1;
private static final int P_24 = 0;

private static final int P_30 = 0;
private static final int P_31 = 1;
```

```
private static final int P_32 = 1;
private static final int P_33 = P_31 ^ 1;
private static final int P_34 = 1;

private static final int GF256_FDBK = 0x169;
private static final int GF256_FDBK_2 = 0x169 / 2;
private static final int GF256_FDBK_4 = 0x169 / 4;

private static final int[][] MDS = new int[4][256]; // blank final

private static final int RS_GF_FDBK = 0x14D; // field generator

private static final char[] HEX_DIGITS = {
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D',
        'E', 'F'
};

static {
    long time = System.currentTimeMillis();

    //
    // precompute the MDS matrix
    //
    int[] m1 = new int[2];
    int[] mX = new int[2];
    int[] mY = new int[2];
    int i, j;
    for (i = 0; i < 256; i++) {
        j = P[0][i] & 0xFF; // compute all the matrix elements
        m1[0] = j;
        mX[0] = Mx_X(j) & 0xFF;
        mY[0] = Mx_Y(j) & 0xFF;

        j = P[1][i] & 0xFF;
        m1[1] = j;
        mX[1] = Mx_X(j) & 0xFF;
        mY[1] = Mx_Y(j) & 0xFF;
```

```
            MDS[0][i] = m1[P_00] << 0
                    | // fill matrix w/ above elements
                    mX[P_00] << 8
                    | mY[P_00] << 16
                    | mY[P_00] << 24;
            MDS[1][i] = mY[P_10] << 0
                    | mY[P_10] << 8
                    | mX[P_10] << 16
                    | m1[P_10] << 24;
            MDS[2][i] = mX[P_20] << 0
                    | mY[P_20] << 8
                    | m1[P_20] << 16
                    | mY[P_20] << 24;
            MDS[3][i] = mX[P_30] << 0
                    | m1[P_30] << 8
                    | mY[P_30] << 16
                    | mX[P_30] << 24;
        }

        time = System.currentTimeMillis() - time;
    }

    private static final int LFSR1(int x) {
        return (x >> 1)
                ^ ((x & 0x01) != 0 ? GF256_FDBK_2 : 0);
    }

    private static final int LFSR2(int x) {
        return (x >> 2)
                ^ ((x & 0x02) != 0 ? GF256_FDBK_2 : 0)
                ^ ((x & 0x01) != 0 ? GF256_FDBK_4 : 0);
    }

    private static final int Mx_1(int x) {
        return x;
    }
```

```
private static final int Mx_X(int x) {
    return x ^ LFSR2(x);
}           // 5B

private static final int Mx_Y(int x) {
    return x ^ LFSR1(x) ^ LFSR2(x);
} // EF

private static final int b0(int x) {
    return x & 0xFF;
}

private static final int b1(int x) {
    return (x >>> 8) & 0xFF;
}

private static final int b2(int x) {
    return (x >>> 16) & 0xFF;
}

private static final int b3(int x) {
    return (x >>> 24) & 0xFF;
}

private static final int RS_MDS_Encode(int k0, int k1) {
    int r = k1;
    for (int i = 0; i < 4; i++) // shift 1 byte at a time
    {
        r = RS_rem(r);
    }
    r ^= k0;
    for (int i = 0; i < 4; i++) {
        r = RS_rem(r);
    }
    return r;
}
```

```
private static final int RS_rem(int x) {
    int b = (x >>> 24) & 0xFF;
    int g2 = ((b << 1) ^ ((b & 0x80) != 0 ? RS_GF_FDBK : 0)) & 0xFF;
    int g3 = (b >>> 1) ^ ((b & 0x01) != 0 ? (RS_GF_FDBK >>> 1) : 0) ^ g2;
    int result = (x << 8) ^ (g3 << 24) ^ (g2 << 16) ^ (g3 << 8) ^ b;
    return result;
}

private static final int F32(int k64Cnt, int x, int[] k32) {
    int b0 = b0(x);
    int b1 = b1(x);
    int b2 = b2(x);
    int b3 = b3(x);
    int k0 = k32[0];
    int k1 = k32[1];
    int k2 = k32[2];
    int k3 = k32[3];

    int result = 0;
    switch (k64Cnt & 3) {
        case 1:
            result
                    = MDS[0][(P[P_01][b0] & 0xFF) ^ b0(k0)]
                    ^ MDS[1][(P[P_11][b1] & 0xFF) ^ b1(k0)]
                    ^ MDS[2][(P[P_21][b2] & 0xFF) ^ b2(k0)]
                    ^ MDS[3][(P[P_31][b3] & 0xFF) ^ b3(k0)];
            break;
        case 0: // same as 4
            b0 = (P[P_04][b0] & 0xFF) ^ b0(k3);
            b1 = (P[P_14][b1] & 0xFF) ^ b1(k3);
            b2 = (P[P_24][b2] & 0xFF) ^ b2(k3);
            b3 = (P[P_34][b3] & 0xFF) ^ b3(k3);
        case 3:
            b0 = (P[P_03][b0] & 0xFF) ^ b0(k2);
            b1 = (P[P_13][b1] & 0xFF) ^ b1(k2);
            b2 = (P[P_23][b2] & 0xFF) ^ b2(k2);
```

```
            b3 = (P[P_33][b3] & 0xFF) ^ b3(k2);
        case 2:                         // 128-bit keys (optimize for this
            case)
            result
                    = MDS[0][(P[P_01][(P[P_02][b0] & 0xFF) ^ b0(k1)] & 0xFF)
                        ^ b0(k0)]
                    ^ MDS[1][(P[P_11][(P[P_12][b1] & 0xFF) ^ b1(k1)] & 0xFF)
                        ^ b1(k0)]
                    ^ MDS[2][(P[P_21][(P[P_22][b2] & 0xFF) ^ b2(k1)] & 0xFF)
                        ^ b2(k0)]
                    ^ MDS[3][(P[P_31][(P[P_32][b3] & 0xFF) ^ b3(k1)] & 0xFF)
                        ^ b3(k0)];
            break;
    }
    return result;
}

private static final int Fe32(int[] sBox, int x, int R) {
    return sBox[2 * _b(x, R)]
            ^ sBox[2 * _b(x, R + 1) + 1]
            ^ sBox[0x200 + 2 * _b(x, R + 2)]
            ^ sBox[0x200 + 2 * _b(x, R + 3) + 1];
}

private static final int _b(int x, int N) {
    int result = 0;
    switch (N % 4) {
        case 0:
            result = b0(x);
            break;
        case 1:
            result = b1(x);
            break;
        case 2:
            result = b2(x);
            break;
        case 3:
```

```
            result = b3(x);
            break;
    }
    return result;
}

public static int blockSize() {
    return BLOCK_SIZE;
}

public static byte[] encrypt(byte[] kb, byte[] pt) {
    try {
        Object key = makeKey(kb);
        byte[] ct = blockEncrypt(pt, 0, key);
        return ct;
    } catch (InvalidKeyException ex) {
        Logger.getLogger(Twofish_Algorithm.class.getName()).log(Level.SEVERE,
            null, ex);
        return null;
    }
}

public static byte[] decrypt(byte[] kb, byte[] ct) {
    try {
        Object key = makeKey(kb);
        byte[] pt = blockDecrypt(ct, 0, key);
        return pt;
    } catch (InvalidKeyException ex) {
        Logger.getLogger(Twofish_Algorithm.class.getName()).log(Level.SEVERE,
            null, ex);
        return null;
    }
}

private static boolean areEqual(byte[] a, byte[] b) {
    int aLength = a.length;
    if (aLength != b.length) {
```

```
        return false;
    }
    for (int i = 0; i < aLength; i++) {
        if (a[i] != b[i]) {
            return false;
        }
    }
    return true;
}

private static String intToString(int n) {
    char[] buf = new char[8];
    for (int i = 7; i >= 0; i--) {
        buf[i] = HEX_DIGITS[n & 0x0F];
        n >>>= 4;
    }
    return new String(buf);
}

private static String toString(byte[] ba) {
    return toString(ba, 0, ba.length);
}

private static String toString(byte[] ba, int offset, int length) {
    char[] buf = new char[length * 2];
    for (int i = offset, j = 0, k; i < offset + length;) {
        k = ba[i++];
        buf[j++] = HEX_DIGITS[(k >>> 4) & 0x0F];
        buf[j++] = HEX_DIGITS[k & 0x0F];
    }
    return new String(buf);
}

 public static byte[]
        blockEncrypt(byte[] in, int inOffset, Object sessionKey) {

    Object[] sk = (Object[]) sessionKey; // extract S-box and session key
```

```
int[] sBox = (int[]) sk[0];
int[] sKey = (int[]) sk[1];

int x0 = (in[inOffset++] & 0xFF)
       | (in[inOffset++] & 0xFF) << 8
       | (in[inOffset++] & 0xFF) << 16
       | (in[inOffset++] & 0xFF) << 24;
int x1 = (in[inOffset++] & 0xFF)
       | (in[inOffset++] & 0xFF) << 8
       | (in[inOffset++] & 0xFF) << 16
       | (in[inOffset++] & 0xFF) << 24;
int x2 = (in[inOffset++] & 0xFF)
       | (in[inOffset++] & 0xFF) << 8
       | (in[inOffset++] & 0xFF) << 16
       | (in[inOffset++] & 0xFF) << 24;
int x3 = (in[inOffset++] & 0xFF)
       | (in[inOffset++] & 0xFF) << 8
       | (in[inOffset++] & 0xFF) << 16
       | (in[inOffset++] & 0xFF) << 24;

int t0, t1;
int k = ROUND_SUBKEYS;

SecureRandom psi = new SecureRandom();

x2 = (x2 ^ sKey[INPUT_WHITEN + 2] ^ Fe32(sBox, x0 ^ sKey[INPUT_WHITEN],
    0) + Fe32(sBox, x1 ^ sKey[INPUT_WHITEN + 1], 3) + sKey[k++]) ^
    psi.hashCode();
x2 = (x2 ^ psi.hashCode()) >>> 1 | (x2 ^ psi.hashCode()) << 31;
x3 = ((x3 ^ sKey[INPUT_WHITEN + 3]) << 1 | (x3 ^ sKey[INPUT_WHITEN + 3])
    >>> 31) ^ Fe32(sBox, x0 ^ sKey[INPUT_WHITEN], 0) + 2 * Fe32(sBox, x1
    ^ sKey[INPUT_WHITEN + 1], 3) + sKey[k++];

x0 = (x0 ^ sKey[INPUT_WHITEN] ^ Fe32(sBox, x2, 0) + Fe32(sBox, x3, 3) +
    sKey[k++]) ^ psi.hashCode();
x0 = (x0 ^ psi.hashCode()) >>> 1 | (x0 ^ psi.hashCode()) << 31;
x1 = ((x1 ^ sKey[INPUT_WHITEN + 1]) << 1 | (x1 ^ sKey[INPUT_WHITEN + 1])
```

```
        >>> 31) ^ Fe32(sBox, x2, 0) + 2 * Fe32(sBox, x3, 3) + sKey[k++];

    for (int R = 2; R < ROUNDS - 2; R += 2) {
        x2 ^= (Fe32(sBox, x0, 0) + Fe32(sBox, x1, 3) + sKey[k++]) ^
            psi.hashCode();
        x2 = (x2 ^ psi.hashCode()) >>> 1 | (x2 ^ psi.hashCode()) << 31;
        x3 = (x3 << 1 | x3 >>> 31) ^ Fe32(sBox, x0, 0) + 2 * Fe32(sBox, x1,
            3) + sKey[k++];

        x0 ^= (Fe32(sBox, x2, 0) + Fe32(sBox, x3, 3) + sKey[k++]) ^
            psi.hashCode();
        x0 = (x0 ^ psi.hashCode()) >>> 1 | (x0 ^ psi.hashCode()) << 31;
        x1 = (x1 << 1 | x1 >>> 31) ^ Fe32(sBox, x2, 0) + 2 * Fe32(sBox, x3,
            3) + sKey[k++];
    }

    x2 ^= (Fe32(sBox, x0, 0) + Fe32(sBox, x1, 3) + sKey[k++]) ^
        psi.hashCode();
    x2 = ((x2 ^ psi.hashCode()) >>> 1 | (x2 ^ psi.hashCode()) << 31) ^
        sKey[OUTPUT_WHITEN];
    x3 = ((x3 << 1 | x3 >>> 31) ^ Fe32(sBox, x0, 0) + 2 * Fe32(sBox, x1, 3)
        + sKey[k++]) ^ sKey[OUTPUT_WHITEN + 1];

    x0 ^= (Fe32(sBox, x2 ^ sKey[OUTPUT_WHITEN], 0) + Fe32(sBox, x3 ^
        sKey[OUTPUT_WHITEN + 1], 3) + sKey[k++]) ^ psi.hashCode();
    x0 = ((x0 ^ psi.hashCode()) >>> 1 | (x0 ^ psi.hashCode()) << 31) ^
        sKey[OUTPUT_WHITEN + 2];
    x1 = ((x1 << 1 | x1 >>> 31) ^ Fe32(sBox, x2 ^ sKey[OUTPUT_WHITEN], 0) +
        2 * Fe32(sBox, x3 ^ sKey[OUTPUT_WHITEN + 1], 3) + sKey[k++]) ^
        sKey[OUTPUT_WHITEN + 3];

    int[] resultInt = new int[]{x2, x3, x0, x1};

    byte[] result = UtilWBTwofish.ConcatByteArrayFromInt(resultInt, 32, 16);

    return result;
}
```

```
public static byte[]
        blockDecrypt(byte[] in, int inOffset, Object sessionKey) {

    Object[] sk = (Object[]) sessionKey; // extract S-box and session key
    int[] sBox = (int[]) sk[0];
    int[] sKey = (int[]) sk[1];

    int x2 = (in[inOffset++] & 0xFF)
           | (in[inOffset++] & 0xFF) << 8
           | (in[inOffset++] & 0xFF) << 16
           | (in[inOffset++] & 0xFF) << 24;
    int x3 = (in[inOffset++] & 0xFF)
           | (in[inOffset++] & 0xFF) << 8
           | (in[inOffset++] & 0xFF) << 16
           | (in[inOffset++] & 0xFF) << 24;
    int x0 = (in[inOffset++] & 0xFF)
           | (in[inOffset++] & 0xFF) << 8
           | (in[inOffset++] & 0xFF) << 16
           | (in[inOffset++] & 0xFF) << 24;
    int x1 = (in[inOffset++] & 0xFF)
           | (in[inOffset++] & 0xFF) << 8
           | (in[inOffset++] & 0xFF) << 16
           | (in[inOffset++] & 0xFF) << 24;

    int k = ROUND_SUBKEYS + 2 * ROUNDS - 1;
    int t0, t1;

    SecureRandom psi = new SecureRandom();

    x1 = (x1 ^ sKey[OUTPUT_WHITEN + 3] ^ Fe32(sBox, x2 ^
        sKey[OUTPUT_WHITEN], 0) + 2 * Fe32(sBox, x3 ^ sKey[OUTPUT_WHITEN +
        1], 3) + sKey[k--]) ^ psi.hashCode();
    x1 = (x1 ^ psi.hashCode()) >>> 1 | (x1 ^ psi.hashCode()) << 31;
    x0 = ((x0 ^ sKey[OUTPUT_WHITEN + 2]) << 1 | (x0 ^ sKey[OUTPUT_WHITEN +
        2]) >>> 31) ^ Fe32(sBox, x2 ^ sKey[OUTPUT_WHITEN], 0) + Fe32(sBox,
        x3 ^ sKey[OUTPUT_WHITEN + 1], 3) + sKey[k--];
```

```
x3 = (x3 ^ sKey[OUTPUT_WHITEN + 1] ^ Fe32(sBox, x0, 0) + 2 * Fe32(sBox,
    x1, 3) + sKey[k--]) ^ psi.hashCode();
x3 = (x3 ^ psi.hashCode()) >>> 1 | (x3 ^ psi.hashCode()) << 31;
x2 = ((x2 ^ sKey[OUTPUT_WHITEN]) << 1 | (x2 ^ sKey[OUTPUT_WHITEN]) >>>
    31) ^ Fe32(sBox, x0, 0) + Fe32(sBox, x1, 3) + sKey[k--];

for (int R = 2; R < ROUNDS - 2; R += 2) {
    x1 ^= (Fe32(sBox, x2, 0) + 2 * Fe32(sBox, x3, 3) + sKey[k--]) ^
        psi.hashCode();
    x1 = (x1 ^ psi.hashCode()) >>> 1 | (x1 ^ psi.hashCode()) << 31;
    x0 = (x0 << 1 | x0 >>> 31) ^ Fe32(sBox, x2, 0) + Fe32(sBox, x3, 3) +
        sKey[k--];

    x3 ^= (Fe32(sBox, x0, 0) + 2 * Fe32(sBox, x1, 3) + sKey[k--]) ^
        psi.hashCode();
    x3 = (x3 ^ psi.hashCode()) >>> 1 | (x3 ^ psi.hashCode()) << 31;
    x2 = (x2 << 1 | x2 >>> 31) ^ Fe32(sBox, x0, 0) + Fe32(sBox, x1, 3) +
        sKey[k--];
}

x1 ^= (Fe32(sBox, x2, 0) + 2 * Fe32(sBox, x3, 3) + sKey[k--]) ^
    psi.hashCode();
x1 = ((x1 ^ psi.hashCode()) >>> 1 | (x1 ^ psi.hashCode()) << 31) ^
    sKey[INPUT_WHITEN + 1];
x0 = ((x0 << 1 | x0 >>> 31) ^ Fe32(sBox, x2, 0) + Fe32(sBox, x3, 3) +
    sKey[k--]) ^ sKey[INPUT_WHITEN];

x3 ^= (Fe32(sBox, x0 ^ sKey[INPUT_WHITEN], 0) + 2 * Fe32(sBox, x1 ^
    sKey[INPUT_WHITEN + 1], 3) + sKey[k--]) ^ psi.hashCode();
x3 = ((x3 ^ psi.hashCode()) >>> 1 | (x3 ^ psi.hashCode()) << 31) ^
    sKey[INPUT_WHITEN + 3];
x2 = ((x2 << 1 | x2 >>> 31) ^ Fe32(sBox, x0 ^ sKey[INPUT_WHITEN], 0) +
    Fe32(sBox, x1 ^ sKey[INPUT_WHITEN + 1], 3) + sKey[k--]) ^
    sKey[INPUT_WHITEN + 2];

int[] resultInt = new int[]{x0, x1, x2, x3};
```

```
        byte[] result = UtilWBTwofish.ConcatByteArrayFromInt(resultInt, 32, 16);

        return result;
    }
```

```
public static synchronized Object makeKey(byte[] k)
        throws InvalidKeyException {
    if (DEBUG) {
        trace(IN, "makeKey(" + k + ")");
    }
    if (k == null) {
        throw new InvalidKeyException("Empty key");
    }
    int length = k.length;
    if (!(length == 8 || length == 16 || length == 24 || length == 32)) {
        throw new InvalidKeyException("Incorrect key length");
    }

    int k64Cnt = length / 8;
    int subkeyCnt = ROUND_SUBKEYS + 2 * ROUNDS;
    int[] k32e = new int[4]; // even 32-bit entities
    int[] k32o = new int[4]; // odd 32-bit entities
    int[] sBoxKey = new int[4];

    int i, j, offset = 0;
    for (i = 0, j = k64Cnt - 1; i < 4 && offset < length; i++, j--) {
        k32e[i] = (k[offset++] & 0xFF)
                | (k[offset++] & 0xFF) << 8
                | (k[offset++] & 0xFF) << 16
                | (k[offset++] & 0xFF) << 24;
        k32o[i] = (k[offset++] & 0xFF)
                | (k[offset++] & 0xFF) << 8
                | (k[offset++] & 0xFF) << 16
                | (k[offset++] & 0xFF) << 24;
        sBoxKey[j] = RS_MDS_Encode(k32e[i], k32o[i]); // reverse order
    }
```

```
int q, A, B;
int[] subKeys = new int[subkeyCnt];
for (i = q = 0; i < subkeyCnt / 2; i++, q += SK_STEP) {
    A = F32(k64Cnt, q, k32e); // A uses even key entities
    B = F32(k64Cnt, q + SK_BUMP, k32o); // B uses odd key entities
    B = B << 8 | B >>> 24;
    A += B;
    subKeys[2 * i] = A;            // combine with a PHT
    A += B;
    subKeys[2 * i + 1] = A << SK_ROTL | A >>> (32 - SK_ROTL);
}

int k0 = sBoxKey[0];
int k1 = sBoxKey[1];
int k2 = sBoxKey[2];
int k3 = sBoxKey[3];
int b0, b1, b2, b3;
int[] sBox = new int[4 * 256];

for (i = 0; i < 256; i++) {
    b0 = b1 = b2 = b3 = i;
    switch (k64Cnt & 3) {
        case 1:
            sBox[2 * i] = MDS[0][(P[P_01][b0] & 0xFF) ^ b0(k0)];
            sBox[2 * i + 1] = MDS[1][(P[P_11][b1] & 0xFF) ^ b1(k0)];
            sBox[0x200 + 2 * i] = MDS[2][(P[P_21][b2] & 0xFF) ^ b2(k0)];
            sBox[0x200 + 2 * i + 1] = MDS[3][(P[P_31][b3] & 0xFF) ^
                b3(k0)];
            break;
        case 0: // same as 4
            b0 = (P[P_04][b0] & 0xFF) ^ b0(k3);
            b1 = (P[P_14][b1] & 0xFF) ^ b1(k3);
            b2 = (P[P_24][b2] & 0xFF) ^ b2(k3);
            b3 = (P[P_34][b3] & 0xFF) ^ b3(k3);
        case 3:
            b0 = (P[P_03][b0] & 0xFF) ^ b0(k2);
            b1 = (P[P_13][b1] & 0xFF) ^ b1(k2);
```

```
                b2 = (P[P_23][b2] & 0xFF) ^ b2(k2);
                b3 = (P[P_33][b3] & 0xFF) ^ b3(k2);
            case 2: // 128-bit keys
                sBox[2 * i] = MDS[0][(P[P_01][(P[P_02][b0] & 0xFF) ^ b0(k1)]
                    & 0xFF) ^ b0(k0)];
                sBox[2 * i + 1] = MDS[1][(P[P_11][(P[P_12][b1] & 0xFF) ^
                    b1(k1)] & 0xFF) ^ b1(k0)];
                sBox[0x200 + 2 * i] = MDS[2][(P[P_21][(P[P_22][b2] & 0xFF) ^
                    b2(k1)] & 0xFF) ^ b2(k0)];
                sBox[0x200 + 2 * i + 1] = MDS[3][(P[P_31][(P[P_32][b3] &
                    0xFF) ^ b3(k1)] & 0xFF) ^ b3(k0)];
        }
    }

    Object sessionKey = new Object[]{sBox, subKeys};

    return sessionKey;
}
```

```
public static synchronized Object makeKey(byte[] k)
        throws InvalidKeyException {
    if (k == null) {
        throw new InvalidKeyException("Empty key");
    }
    int length = k.length;
    if (!(length == 8 || length == 16 || length == 24 || length == 32)) {
        throw new InvalidKeyException("Incorrect key length");
    }

    int k64Cnt = length / 8;
    int subkeyCnt = ROUND_SUBKEYS + 2 * ROUNDS;
    int[] k32e = new int[4]; // even 32-bit entities
    int[] k32o = new int[4]; // odd 32-bit entities
    int[] sBoxKey = new int[4];
    //
    // split user key material into even and odd 32-bit entities and
    // compute S-box keys using (12, 8) Reed-Solomon code over GF(256)
    //
    int i, j, offset = 0;
    for (i = 0, j = k64Cnt - 1; i < 4 && offset < length; i++, j--) {
        k32e[i] = (k[offset++] & 0xFF)
                | (k[offset++] & 0xFF) << 8
                | (k[offset++] & 0xFF) << 16
                | (k[offset++] & 0xFF) << 24;
        k32o[i] = (k[offset++] & 0xFF)
                | (k[offset++] & 0xFF) << 8
                | (k[offset++] & 0xFF) << 16
                | (k[offset++] & 0xFF) << 24;
        sBoxKey[j] = RS_MDS_Encode(k32e[i], k32o[i]); // reverse order
    }
```

```
// compute the round decryption subkeys for PHT. these same subkeys
// will be used in encryption but will be applied in reverse order.
int q, A, B;
int[] subKeys = new int[subkeyCnt];
byte[] tempOutput = null;
for (int l = 0; l < subkeyCnt; l++) {
    int tempOffSet = 0;

    if(l==0)
        tempOutput = SCrypt.generate(k, "WBTWofishPlusalt".getBytes(),
            16384, 8, 1, 4);
    else{
        byte[] concatArray = new byte[k.length + tempOutput.length];
        System.arraycopy(k, 0, concatArray, 0, k.length);
        System.arraycopy(tempOutput, 0, concatArray, k.length,
            tempOutput.length);
        tempOutput = SCrypt.generate(concatArray,
            "WBTWofishPlusalt".getBytes(), 16384, 8, 1, 4);
    }
    subKeys[l] =(tempOutput[tempOffSet++] & 0xFF)|
        (tempOutput[tempOffSet++] & 0xFF) << 8
            | (tempOutput[tempOffSet++] & 0xFF) << 16|
                (tempOutput[tempOffSet++] & 0xFF) << 24;
}

// fully expand the table for speed
//
int k0 = sBoxKey[0];
int k1 = sBoxKey[1];
int k2 = sBoxKey[2];
int k3 = sBoxKey[3];
int b0, b1, b2, b3;
int[] sBox = new int[4 * 256];

int[] resultInt = new int[]{k2, k3, k0, k1};
int size = 16;
```

```
        byte[] roundKey = UtilWBTwofish.ConcatByteArrayFromInt(resultInt, 32,
            16);
        byte[] roundKeysForSboxes =
            UtilWBTwofish.createRoundKeysForSboxes(roundKey, size);

        for (int r = 0; r < roundKeysForSboxes.length / size - 1; r++) {

            byte[] newRoundKey = new byte[size];
            System.arraycopy(roundKeysForSboxes, size * r, newRoundKey, 0, size);
            byte[] key_bytesForSboxes =
                UtilWBTwofish.keyBytesDerivation(newRoundKey);

            for (i = 0; i < 256; i++) {

                sBox[2 * i] = UtilWBTwofish.sboxgen(0, 13, i, key_bytesForSboxes);
                sBox[2 * i + 1] = UtilWBTwofish.sboxgen(1, 13, i,
                    key_bytesForSboxes);
                sBox[0x200 + 2 * i] = UtilWBTwofish.sboxgen(2, 13, i,
                    key_bytesForSboxes);
                sBox[0x200 + 2 * i + 1] = UtilWBTwofish.sboxgen(3, 13, i,
                    key_bytesForSboxes);
            }

        }

        Object sessionKey = new Object[]{sBox, subKeys};

        return sessionKey;
    }
} // FIM - Twofish_Algorithm.java
```